Lecture 4
Sequences and iterators

EECS-311

Memory structure

- Ultimately, the computer's memory is **one big array of bytes**
- Objects are represented by discrete **chunks** of the array
  - **Identified by the location** (address) within the array
- **Memory management** is the process of keeping track of
  - **Which chunks are used** to represent which objects
  - **Which chunks are free** (available to allocation)

Statically allocated structures

- Memory management makes it **easy to allocate chunks**
  - But **hard to grow or shrink** them after allocation
- So all data structures are ultimately built out of **static-sized chunks**

- **Arrays**
  - Fast
  - Let you access elements by number
  - Can't be resized
- **Record structures** (classes, structs)
  - Fields accessed by name rather than number
  - Set of fields is fixed at compile time
  - Compiler effectively turns field references into array references

Performance profile of arrays

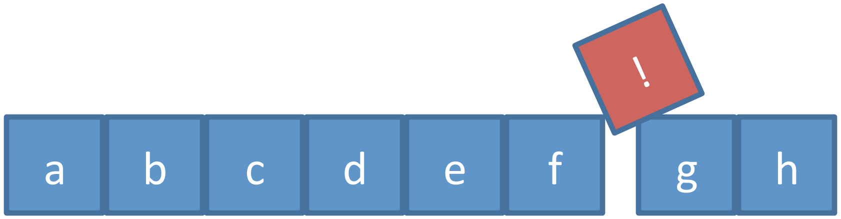
How fast are basic array operations? (n=array size)

- Create (and initialize) array: $\mathcal{O}(n)$
- Access an element: $\mathcal{O}(1)$
- Mutate (modify) an element: $\mathcal{O}(1)$
- Find position of an element: $\mathcal{O}(n)$
- Add or remove element: *impossible*

```
int[] a;
```

Dynamic structures

- If we want to add or remove elements, we have to be fancier



- **Indirection**
  - Store data in a normal array
  - Represent sequence object as a pointer to the array
  - Replace it with a whole new array when you need to change size
- **Chained** structures
  - Break structure into chunks
  - Each chunk points to next chunk
  - Resize by adding or removing chunks
  - Examples
    - Linked lists
    - Files on disk

Collection classes

- Called “dynamic sets” in CLR
- Store a collection of **objects associated with “keys”** used to access them
  - For **arrays**, the keys are **indices** into the array
  - For “**dictionaries**”, the keys can be **arbitrary objects**

Dynamic sequences

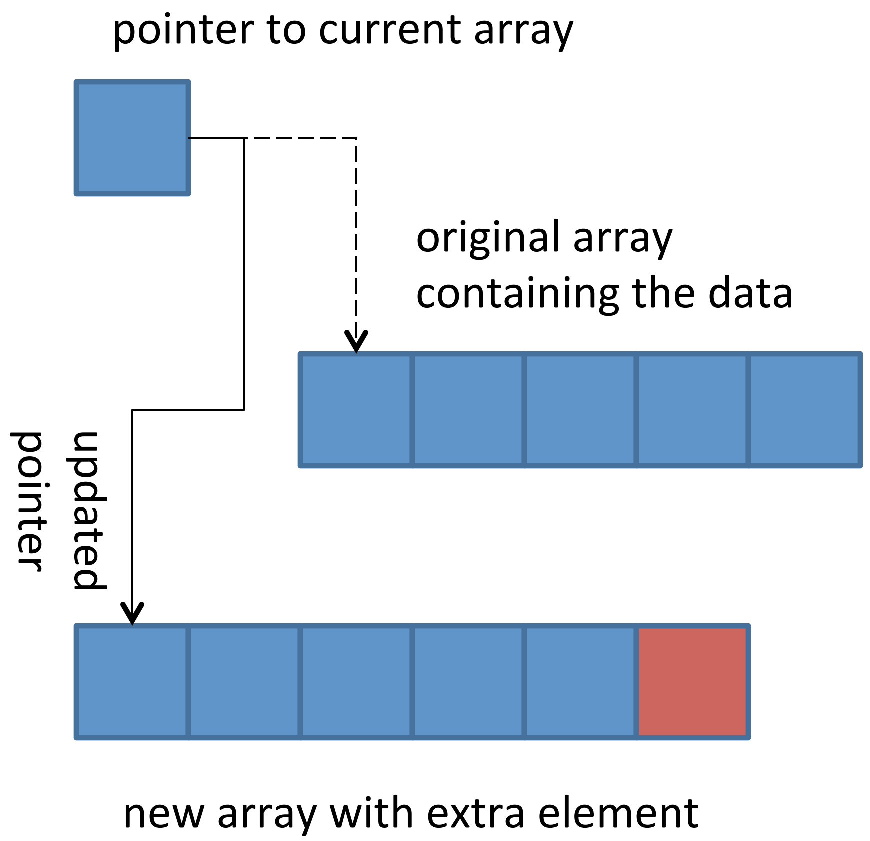
- **Ordered** collection of objects
  - Objects are stored in a definite order, as in arrays
- Can **add or remove** objects
- Vary by **restrictions** on what elements can
  - Be **accessed**
  - Be **added or removed**

Lists (aka sequences)

- **Generalization of arrays**
  - Essentially a mapping from integers (positions in the array) to objects of some type
- No restrictions on what elements can be accessed
- Generally no restrictions on where elements can be added or removed

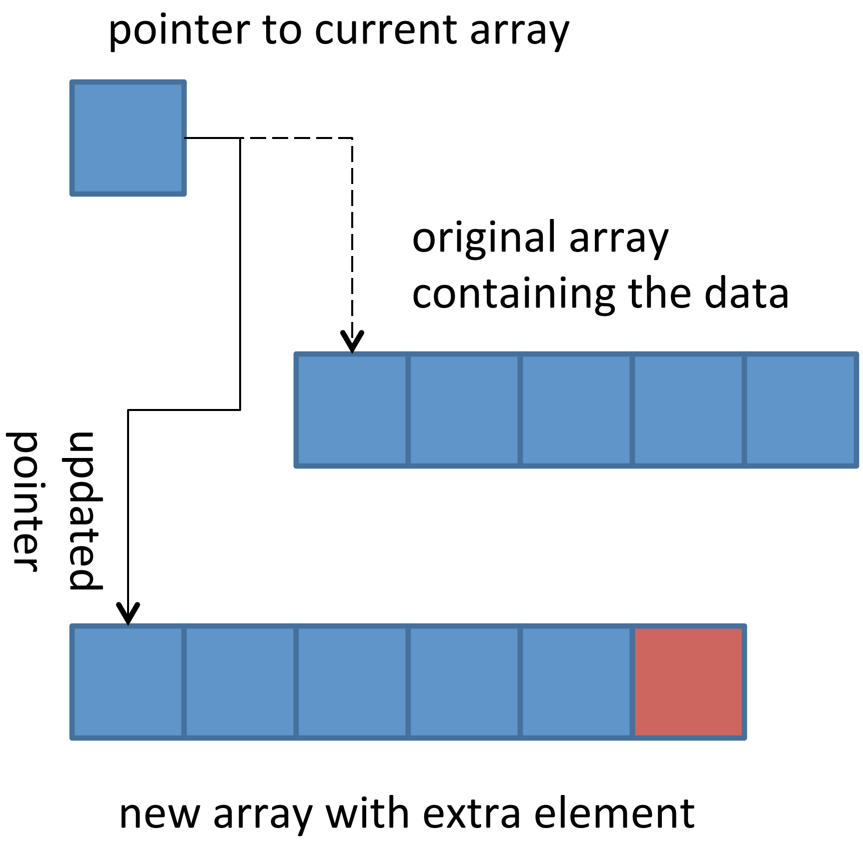
Dynamic arrays

- Just an **object pointing to an array**
- The the real **data is in the array**
- When you need to change the size
  - Make a **new array**
  - **Copy** the data
  - **Change the pointer**


pointer to current array
original array containing the data
updated pointer
new array with extra element

Dynamic arrays

- Element access: $\mathcal{O}(1)$
- Element mutation: $\mathcal{O}(1)$
- Element insertion/ deletion: $\mathcal{O}(n)$


pointer to current array
original array containing the data
updated pointer
new array with extra element

Dynamic array in C#

```
public class DynamicArray
  {
      object[] realArray
                   = new object[0];

      // This overload the [ ] operator
      public object this[int index]
      {
          get
          {
              return realArray[index];
          }
          set
          {
              realArray[index] = value;
          }
      }

      // Add an element at position
      void Add(int position,
                 object newValue)
      {
         object[] newArray
           = new object[realArray.Length + 1];
         for (int i = 0; i < position; i++)
             newArray[i] = realArray[i];
         newArray[position] = newValue;
         for (int i = position+1;
               i<newArray.Length; i++)
             newArray[i] = realArray[i-1];
         realArray = newArray;
      }
      // Removing an element is similar
      // but won’t fit on the slide
  }
```

Iterators

- Provide a **standard interface** for **iterating over items** in a data structure
  - In C# they're used as the interface to foreach
- Specify
  - Where to **start**
  - How to move to the **next element**
  - How to know when you're **done**

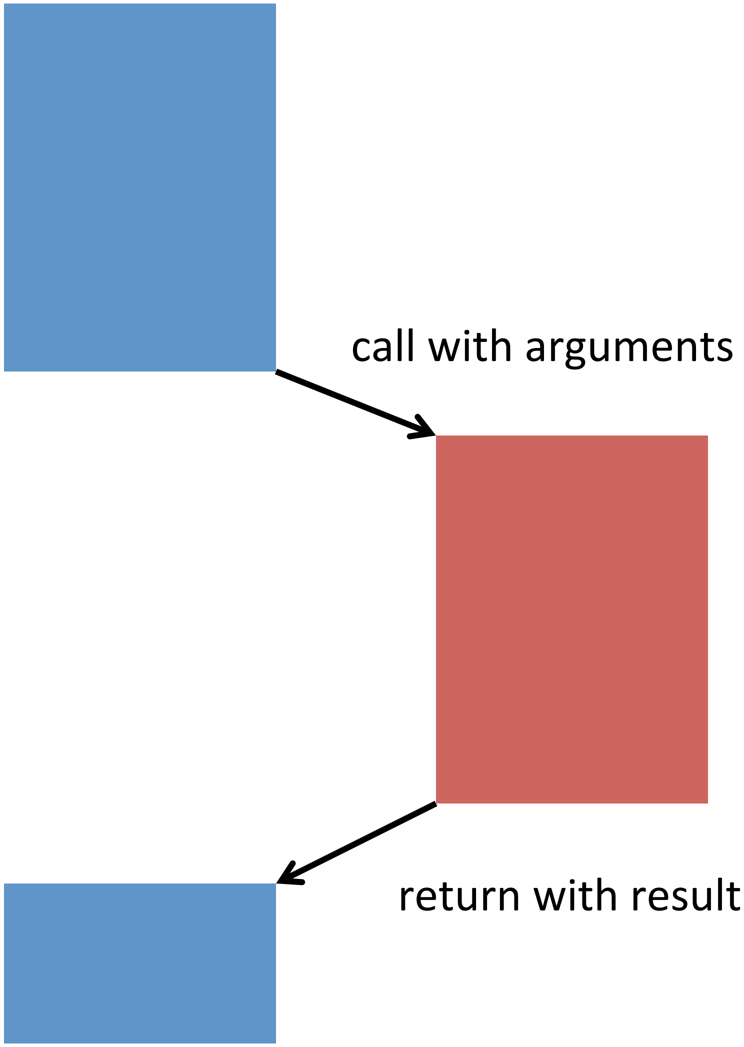
```
foreach (var e in collection) {
    do_something(e);
}
```

Subroutines

- You all know what subroutines are
  - A.k.a **procedures** or **functions**
  - **Methods** are a special case

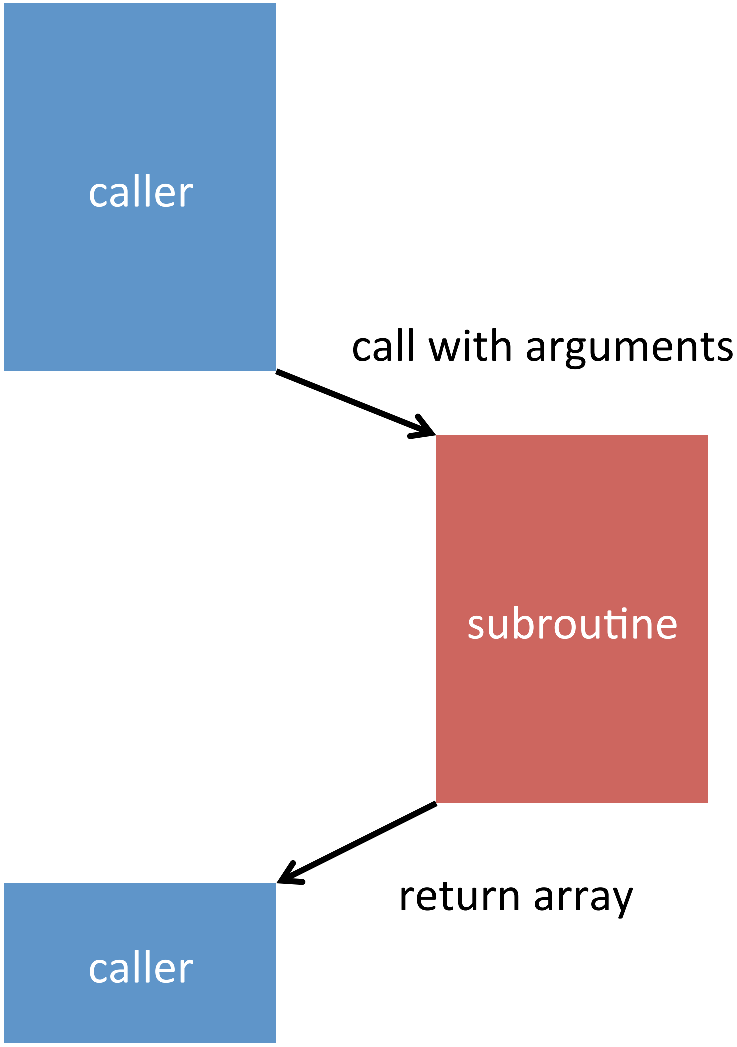
- They're pieces of code that perform a service for other pieces of code

- Control is passed
  - First from the **caller to the subroutine**
  - Them **back to the caller**, along with the return value
  - After which, the **call is done**



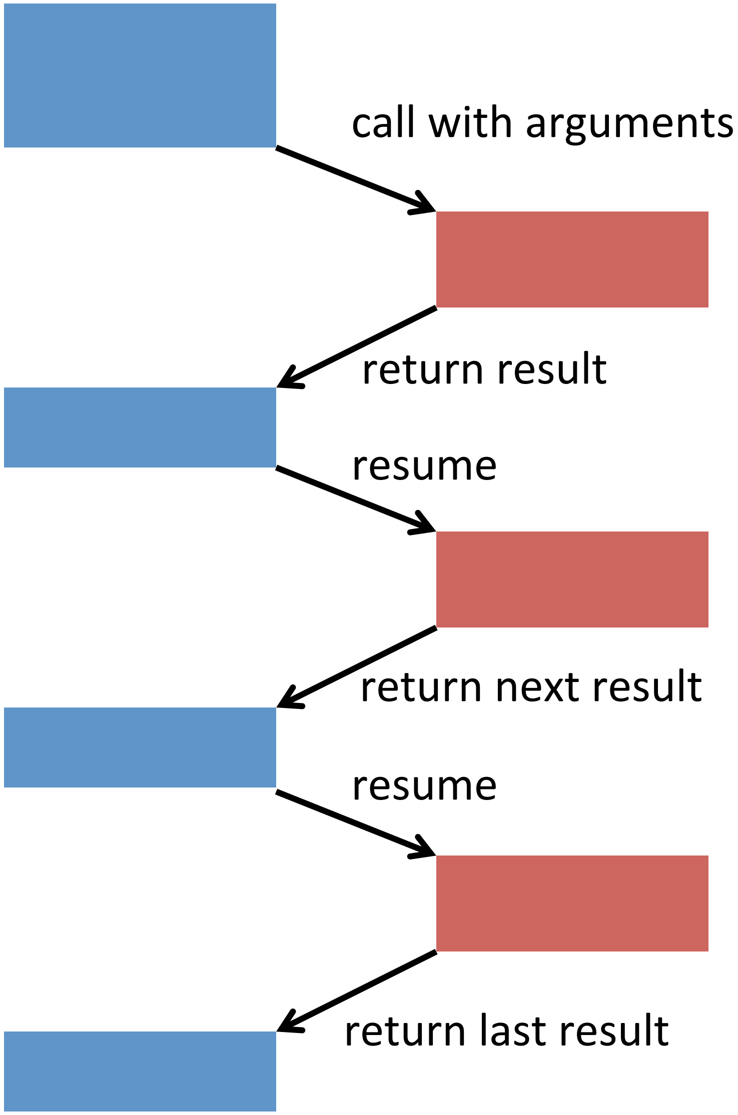
Producer-consumer relationships

- What if you want to return **multiple values**?
- Could return the values in an array or other sequence structure
  - But that's inefficient because you need to allocate new memory
- What you wish is that the subroutine could **"return" multiple times**
  - Once per result



Coroutines

- **Coroutines** are a way of letting a procedure return multiple values
- **Remember their place** (and local variables) when they return
- Can be **resumed** by caller to request another value
  - Coroutine then continues where it left off
  - Computes another value
  - And returns it, again remembering its place
- Eventually, there's usually a way of signaling **completion** of the coroutine
  - I.e. it returns for the last time



call with arguments

return result

resume

return next result

resume

return last result

Iterators in C#

Iterator coroutines in C#

- Must return the magic type **IEnumerable**
- Called using foreach
  - Return values by saying "**yield return** *value*"
  - **Continue where they left off** when when foreach requests the next value
  - Continues until the coroutine exits
    - I.e. it hits the last }

```
public IEnumerable
   IEnumerable.GetEnumerator() {
      loop over all the elements
         yield return current-element;
   }
```

Iterator for dynamic arrays

```
public IEnumerator IEnumerable.GetEnumerator()
{
   for (int i=0; i<realArray.Length; i++)
      yield return realArray[i];
}
```

Now you can use a foreach statement to loop over a DynamicArray

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
      with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d =   { 2, 4, 6, 8 }

sum = 0

```
class DynamicArray : IEnumerable
{
   public IEnumerator
            IEnumerable.GetEnumerator()
    {
      for (int i=0;
             i<realArray.Length;
             i++)
         yield return realArray[i];
    }
    … rest of the members …
}
```

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …


  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

d = { 2, 4, 6, 8 }

sum = 0

```
class DynamicArray : IEnumerable
{
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
    for (int i=0;
           i<realArray.Length;
           i++)
       yield return realArray[i];
  }
  … rest of the members …
}
```

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …


  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

d = { 2, 4, 6, 8 }

sum = 0

```
class DynamicArray : IEnumerable {
  public IEnumerator
         IEnumerable.GetEnumerator()
  {
    for (int i=0;
         i<realArray.Length;
         i++)
      yield return realArray[i];
  }
}
```

i = 0

realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …


  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
    for (int i=0;
         i<realArray.Length;
         i++)
       yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }

sum = 0

i = 0

realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
           IEnumerable.GetEnumerator()
   {
      for (int i=0;
             i<realArray.Length;
             i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 0

i = 0

realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
     for (int i=0;
          i<realArray.Length;
          i++)
        yield return realArray[i];
  }
}
```

d =   { 2, 4, 6, 8 }
sum = 0
e =   2

i =   0
realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …


  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
        IEnumerable.GetEnumerator()
  {
    for (int i=0;
         i<realArray.Length;
         i++)
       yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }

sum = 2

e = 2

i = 0

realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
            IEnumerable.GetEnumerator()
   {
      for (int i=0;
           i<realArray.Length;
           i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 2

e = 2

i = 0

realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
           IEnumerable.GetEnumerator()
  {
     for (int i=0;
           i<realArray.Length;
           i++)
        yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }
sum = 2
e = 2

i = 0
realArray[i] = 2

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
          IEnumerable.GetEnumerator()
   {
      for (int i=0;
           i<realArray.Length;
           i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 2

e = 2

i = 1

realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
        with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
           IEnumerable.GetEnumerator()
   {
     for (int i=0;
           i<realArray.Length;
           i++)
        yield return realArray[i];
   }
}
```

d =   { 2, 4, 6, 8 }

sum = 2

e =   2

i =   1

realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
           IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }
sum = 2
e = 2

i = 1
realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
      with integers …

   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
     for (int i=0;
          i<realArray.Length;
          i++)
        yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }
sum = 2
e = 4

i = 1
realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …

   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d = { 2, 4, 6, 8 }
sum = 6
e = 4

```
class DynamicArray : IEnumerable {
   public IEnumerator
            IEnumerable.GetEnumerator()
   {
      for (int i=0;
           i<realArray.Length;
           i++)
         yield return realArray[i];
   }
}
```

i = 1
realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
            IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 4

i = 1

realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …


  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
    for (int i=0;
          i<realArray.Length;
          i++)
        yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 4

i = 1

realArray[i] = 4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
           IEnumerable.GetEnumerator()
   {
     for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 4

i = 2

realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
          IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 4

i = 2

realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 4

```
class DynamicArray : IEnumerable {
   public IEnumerator
             IEnumerable.GetEnumerator()
   {
      for (int i=0;
             i<realArray.Length;
             i++)
         yield return realArray[i];
   }
}
```

i = 2

realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
      with integers …

  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
    for (int i=0;
         i<realArray.Length;
         i++)
       yield return realArray[i];
  }
}
```

d = { 2, 4, 6, 8 }

sum = 6

e = 6

i = 2

realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
      with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d = { 2, 4, 6, 8 }
sum = 12
e = 6

```
class DynamicArray : IEnumerable {
   public IEnumerator
          IEnumerable.GetEnumerator()
   {
      for (int i=0;
           i<realArray.Length;
           i++)
         yield return realArray[i];
   }
}
```

i = 2
realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
            IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d = { 2, 4, 6, 8 }
sum = 12
e = 6

i = 2
realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d = { 2, 4, 6, 8 }
sum = 12
e = 6

```
class DynamicArray : IEnumerable {
   public IEnumerator
           IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

i = 3
realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d =   { 2, 4, 6, 8 }

sum = 6

e =   4

```
class DynamicArray : IEnumerable {
  public IEnumerator
           IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

i =   3

realArray[i] = 6

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
       with integers …


   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
   public IEnumerator
           IEnumerable.GetEnumerator()
   {
      for (int i=0;
            i<realArray.Length;
            i++)
         yield return realArray[i];
   }
}
```

d =   { 2, 4, 6, 8 }

sum = 6

e =   4

Iterating over a DynamicArray

```
void Main() {
   DynamicArray d;
   … make d and fill it
      with integers …

   int sum = 0;
   foreach (object e in d)
      sum += (int)e;
}
```

d =   { 2, 4, 6, 8 }
sum = 6

```
class DynamicArray : IEnumerable {
   public IEnumerator
            IEnumerable.GetEnumerator()
    {
      for (int i=0;
            i<realArray.Length;
            i++)
          yield return realArray[i];
    }
}
```

Iterating over a DynamicArray

```
void Main() {
  DynamicArray d;
  … make d and fill it
     with integers …

  int sum = 0;
  foreach (object e in d)
    sum += (int)e;
}
```

```
class DynamicArray : IEnumerable {
  public IEnumerator
          IEnumerable.GetEnumerator()
  {
    for (int i=0;
           i<realArray.Length;
           i++)
       yield return realArray[i];
  }
}
```

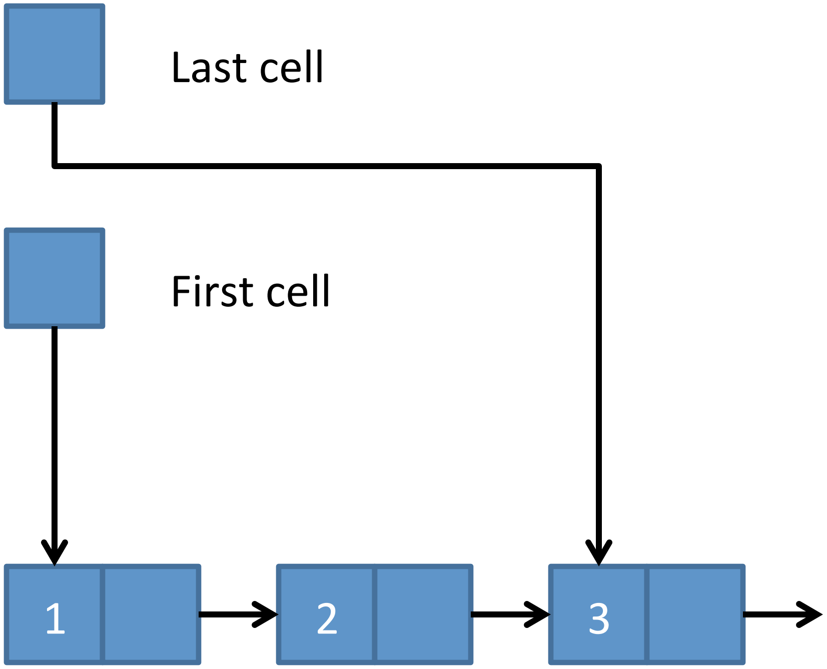
d =   { 2, 4, 6, 8 }
sum = 6

Linked lists

- **Break list into small objects**, one per element

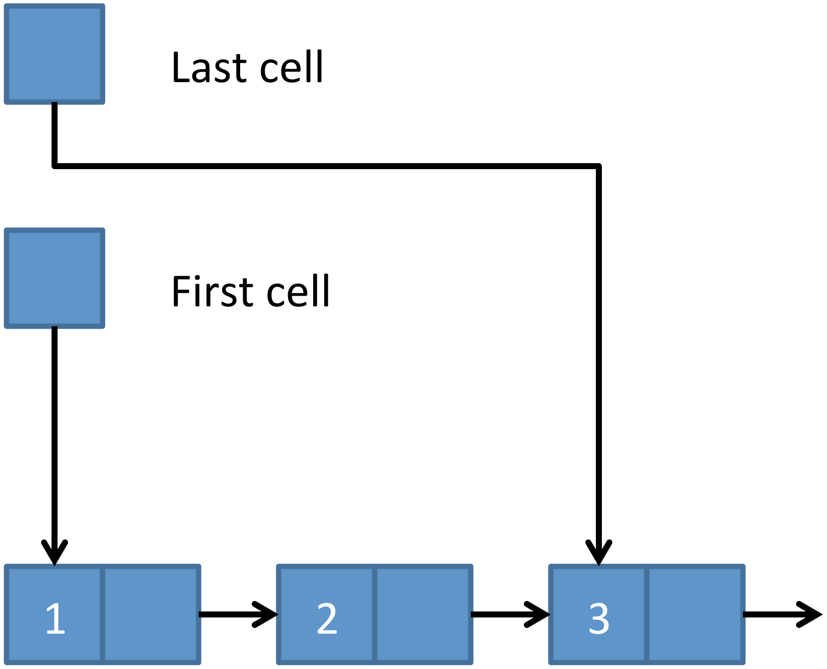
- Each object stores
  - **Value** of one element
  - Pointer to **next object**

- Optional
  - Keep pointers to the first, and/or last cells



Linked lists

- Element access or mutation
  - By element number: $\mathcal{O}(n)$
  - If you already have a pointer to the cell: $\mathcal{O}(1)$
- Element insertion or deletion
  - At beginning: $\mathcal{O}(1)$
  - By element number: $\mathcal{O}(n)$



Linked list in C#

- Single field
  - Points to **first cell**

Note: This uses a feature of C# we haven’t mentioned

- You can put a class inside a class
- And then it’s only visible to the enclosing class

```
class LinkedList : IEnumerable {
    LinkedListCell first;

    … other members …

    class LinkedListCell
    {
        public object value;
        public LinkedListCell next;
    }
}
```

Linked list in C#

- Code for finding an element of the list given index

- Property **this[index]**
  - Overloads the [ ] operator so that you can say, e.g.:

    list[7] = list[8];

```
object this[int index]
{
    get
    {
        LinkedListCell c = first;
        for (int i = 0; i < index; i++)
            c = c.next;
        return c.value;
    }
    set
    {
        LinkedListCell c = first;
        for (int i = 0; i < index; i++)
            c = c.next;
        c.value = value;
    }
}
```

Linked list in C#

- Adding an element at beginning of list is easy

```
void InsertBeginning(object value)
{
    first = new LinkedListCell()
              {
                 value = value,
                 next = first
              };
}
```

Note: This uses another fancy C# feature:

- Putting values of fields in constructor call
- Wrapped in curly braces

Linked list in C#

- Many applications **manipulate list cells directly**

- If you've **already found a cell**, it's easy to insert a new cell after it

```
void InsertAfter(LinkedListCell c,
                                object value) {
   c.next = new LinkedListCell()
                               { value = value,
                                 next = c.next};
}
```

Linked list in C#

- Unfortunately, it’s **hard to insert a cell before** it
  - Because you need to update the next pointer of the **previous cell**
  - And you don’t know what that cell is

```
void InsertAfter(LinkedListCell c,
                         object value) {
  c.next = new LinkedListCell()
                        { value = value,
                          next = c.next};
}
```

Linked list in C#

- Adding a **general element** is ugly
  - Special-case **insertion at beginning**
  - Otherwise search for element **before** the place we're inserting
  - Insert after that

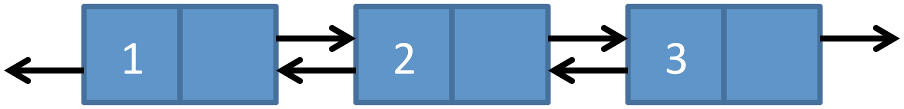
```
void Insert(int position, object value) {
  if (position == 0)
    InsertBeginning(value);
  else {
    LinkedListCell c = first;
    for (int i = 0; i < position-1; i++)
      c = c.next;
    InsertAfter(c);
}
```

Iterator for LinkedLists

```
public IEnumerator GetEnumerator() {
    LinkedListCell c = first;
    while (c != null) {
        yield return c.value;
        c = c.next;
    }
}
```

Doubly linked lists

- Linked lists make it easy to find the cell **after** a given cell but **not before**
- By adding a **second pointer** to the cell that points to the **previous cell**, we can make it easy to
  - Move **forward** and **backward**
  - **Insert elements before** a given cell (not just after)



Doubly-linked list insertion

(note: this is a popular interview question)

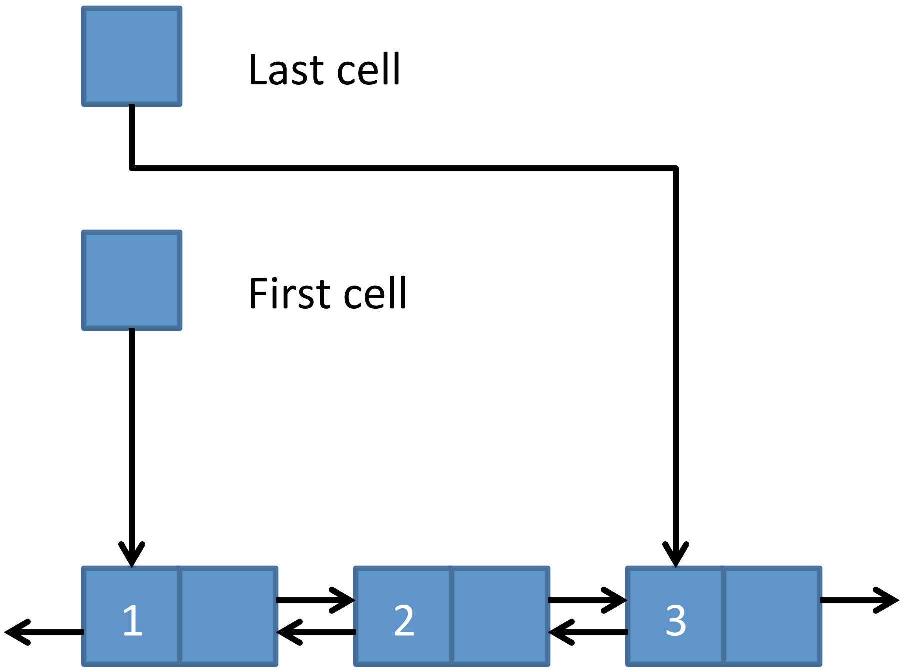
```
class DLLCell
{
    public object value;
    public DLLCell prev;
    public DLLCell next;
}

void InsertBefore(DLLCell c,  object newValue)
{
   InsertBetween(c.prev, c, newValue);
}

void InsertAfter(DLLCell c, object newValue) {
    InsertBetween(c, c.next, newValue);
}
```

```
void InsertBetween(DLLCell before,
                              DLLCell after,
                              object newValue)
{
   DLLCell newCell = new DLLCell()
                               { value = newValue,
                                 prev = before,
                                 next = after };

   before.next = after.prev = newCell;
}
```

Doubly linked lists

- Element access or mutation
  - By element number:
  - If you already have a pointer to the cell:

- Element insertion or deletion
  - At beginning, end, or any other cell you already have a pointer to:
  - By element number:



The .NET list interfaces

- .NET and C# provide two built-in interfaces for the **List abstract data type**

- **IList** is a list of **arbitrary objects**
  - i.e. can contain any kind of data

- **IList<T>** is an IList specialized only contain **data of type T**

IList interface methods

- **int Add(object newValue)**
  Adds an item to the IList. Returns the position where it was added

- **void Clear()**
  Removes all items from the IList.

- **bool Contains(object value)**
  Determines whether the IList contains a specific value.

- **void CopyTo(object[] array, int position)**
  Copies the elements of the IList to an array, starting at a particular position in the destination array

- **IEnumerator GetEnumerator()**
  Returns an enumerator that iterates through a collection.

- **int IndexOf(object value)**
  Determines the index of a specific item in the IList.

- **void Insert(int position, object newValue)**
  Inserts an item to the IList at the specified position.

- **void Remove(object value)**
  Removes the first occurrence of a specific object from the IList.

- **void RemoveAt(int position)**
  Removes the IList item at the specified index.

IList interface properties

- **int Count { get; }**
  Returns the number of items in the list

- **bool IsFixedSize { get; }**
  Tells whether elements can be added and deleted from the list.

- **object this[int index] { get; set; }**
  Overloads the [ ] operator

Generic versions

- .NET also includes typed versions of most of its collection-related interfaces

- So you can use them to define a DynamicArray<T> class, if you like

- IEnumerator<T>
- IEnumerable<T>
- IList<T>
- IStack<T>
- IQueue<T>

Stacks

- Stacks are a **special kind of sequence**

- Addition and deletion are **restricted to the beginning** of the sequence
  - Or you can think of it as restricted to the end, it doesn't really make any difference

- So stacks can be implemented using any data structure for implementing sequences

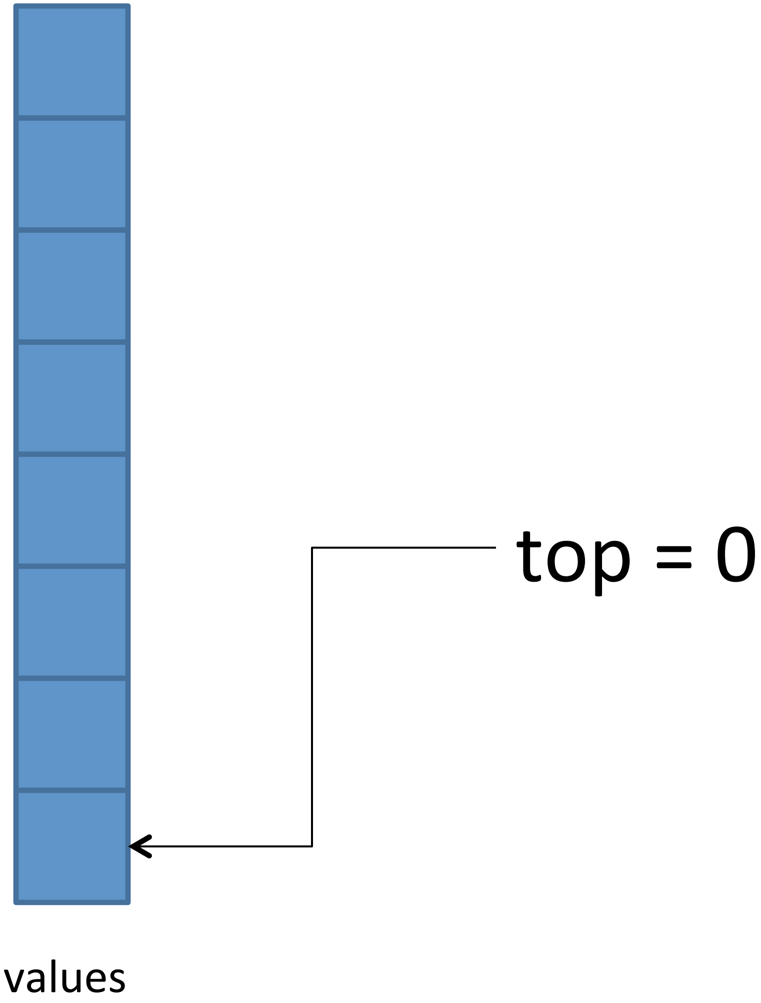
From flickr user matthewpiatt

Implementing a stack with an array

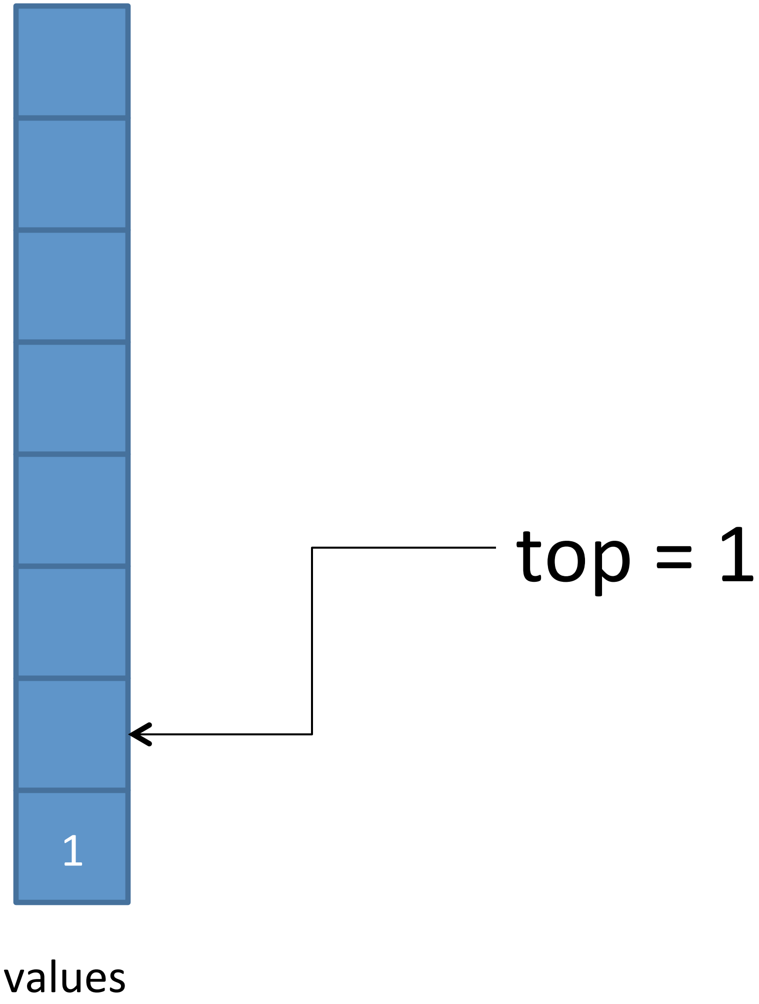
```
class ArrayStack {
   object[] values = new object[100];
   int top = 0;

   void Push(object v) {
      values[top++] = v;
   }
   object Pop() {
      return values[--top];
   }
   bool IsEmpty {
       get { return top==0; }
   }
}
```

Implementing a stack with an array

var s = new ArrayStack();

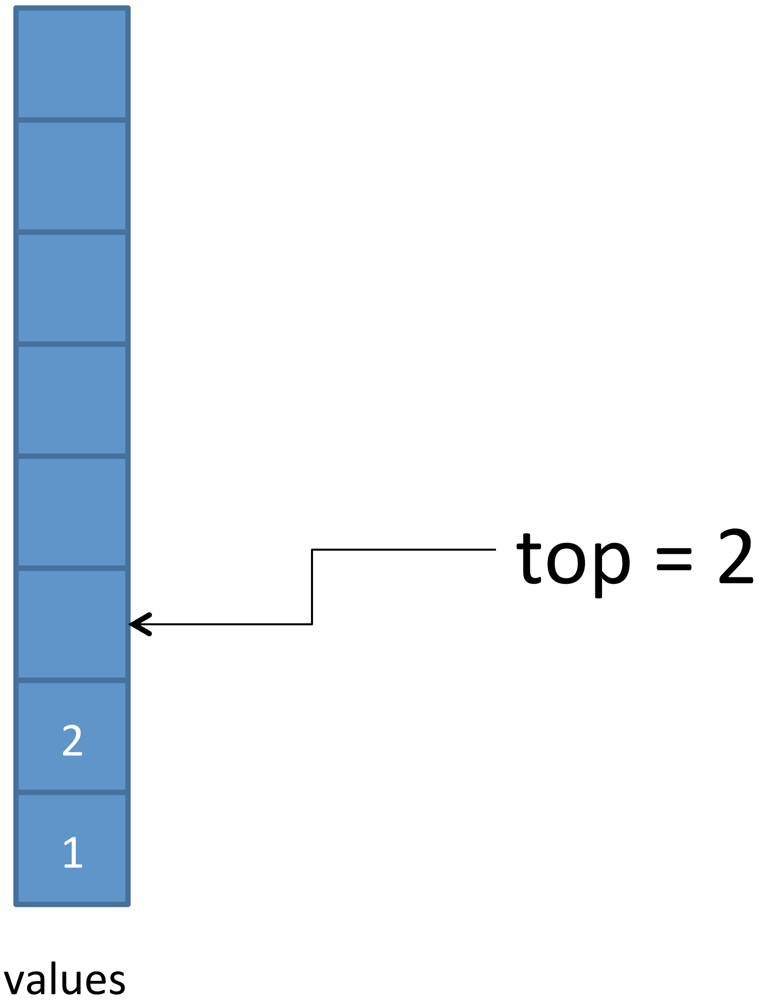


Implementing a stack with an array



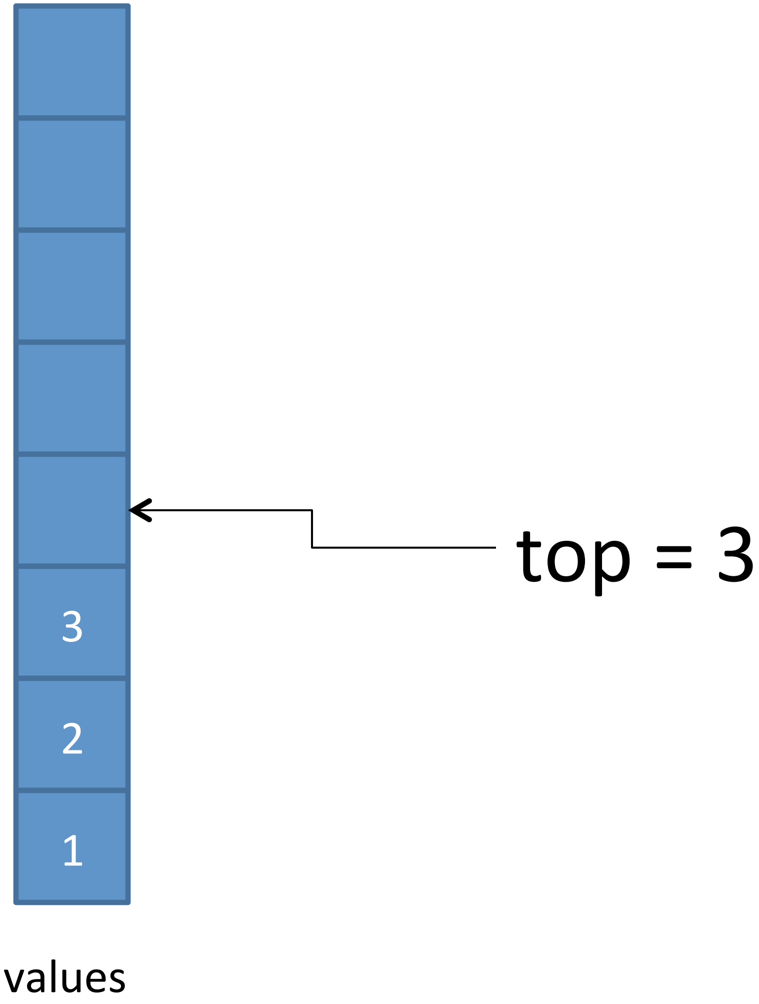
```
var s = new ArrayStack();
s.Push(1);
```

Implementing a stack with an array



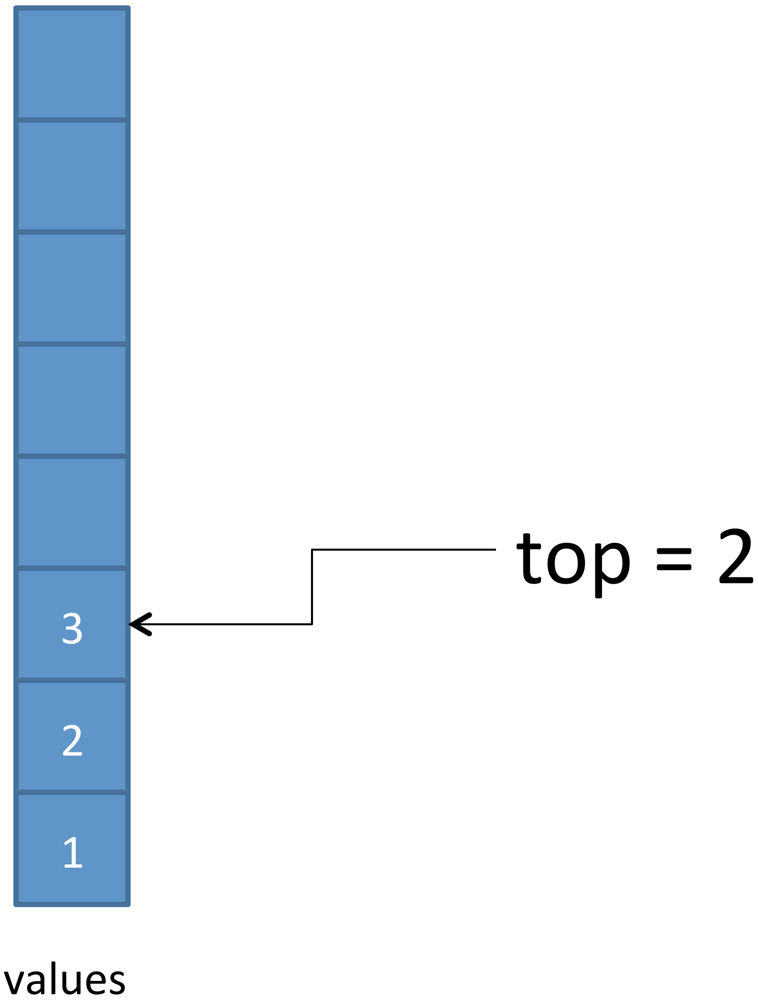
```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
```

Implementing a stack with an array



```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
s.Push(3);
```

Implementing a stack with an array

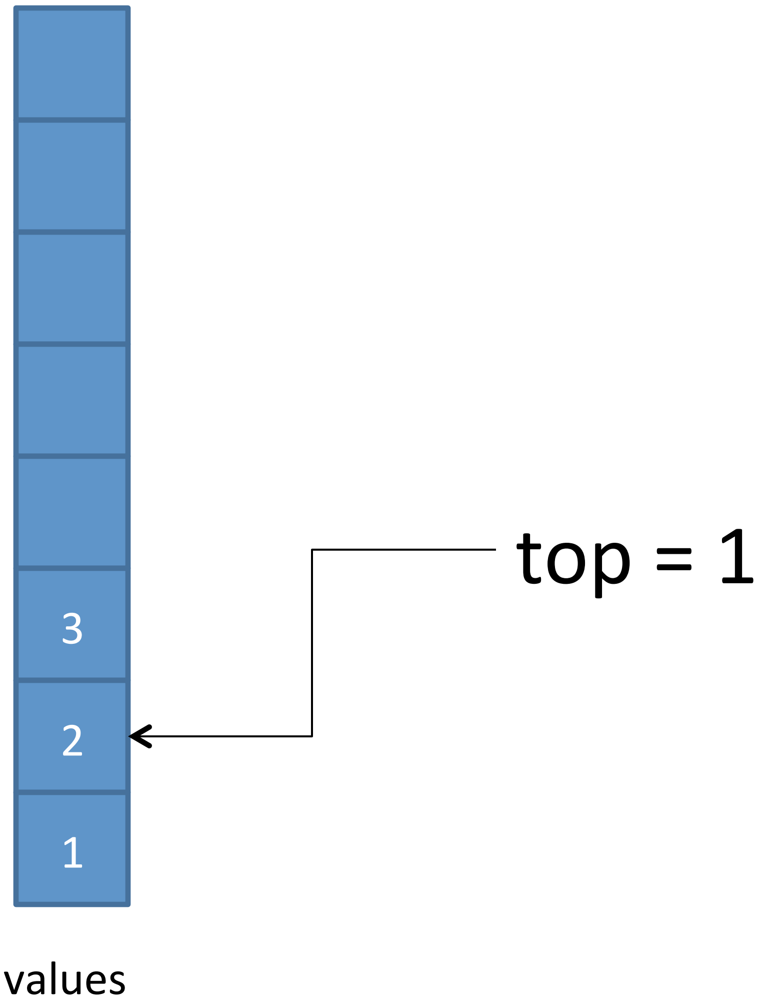


top = 2

values

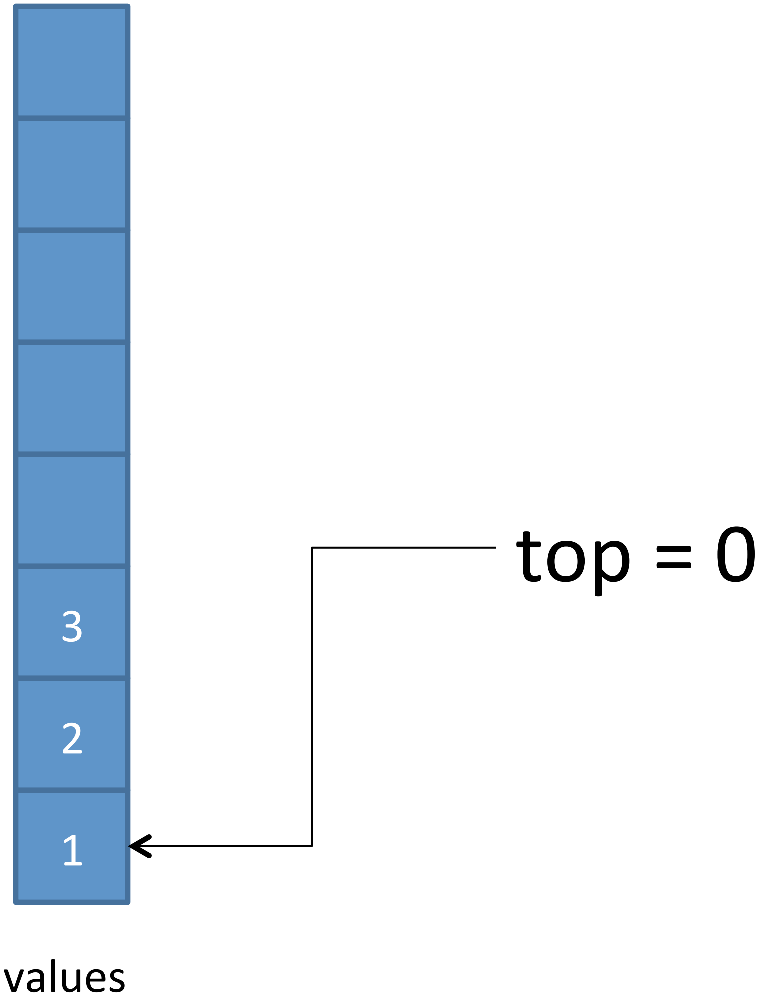
```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop()           // returns 3
```

Implementing a stack with an array



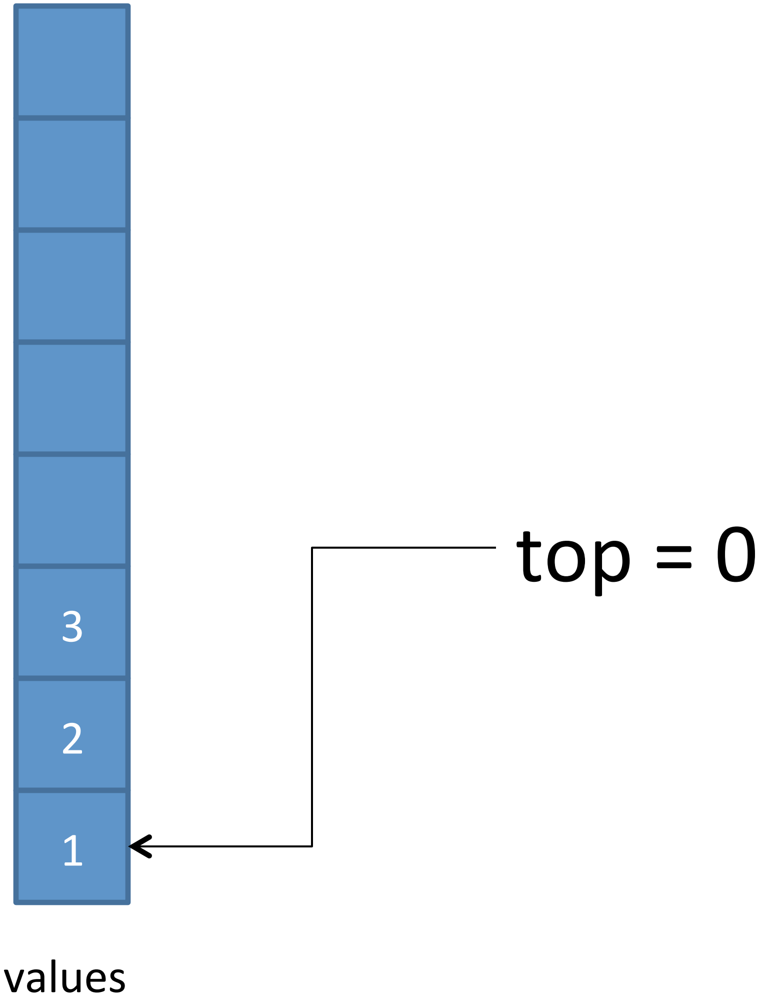
```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop()          // returns 3
s.Pop()          // returns 2
```

Implementing a stack with an array



```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop()         // returns 3
s.Pop()         // returns 2
s.Pop()         // returns 1
```

Implementing a stack with an array



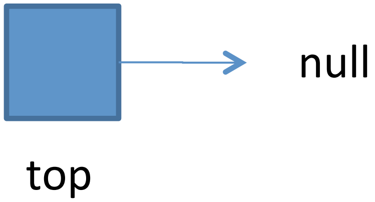
```
var s = new ArrayStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop()         // returns 3
s.Pop()         // returns 2
s.Pop()         // returns 1
s.IsEmpty       // true
```

Implementing a stack with a linked list

```
public class LLStack {
   LLCell top = null;

   void Push(object v) {
      top = new LLCell(v, top);
   }

   object Pop() {
      LLCell oldTop = top;
      top = top.next;
      return oldTop.value;
   }

   bool IsEmpty {
       get { top==null; }
   }
}
```

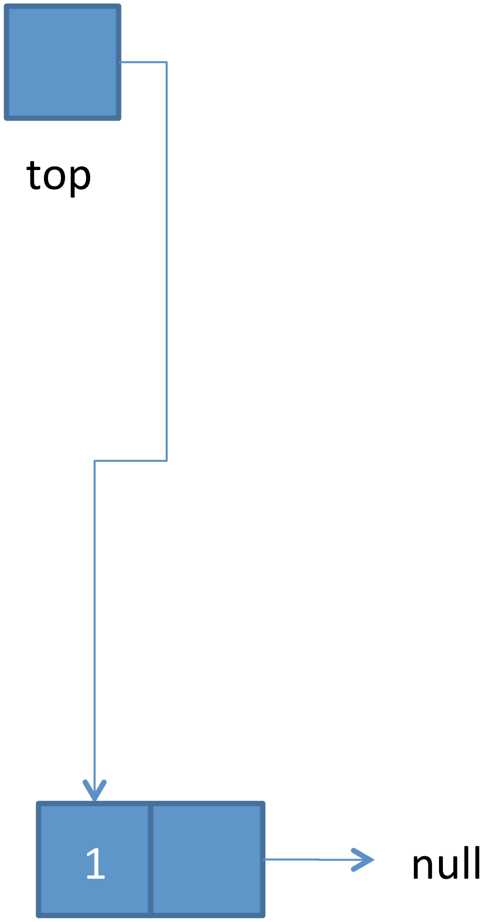
```
public class LLCell {
   public object value
   public LLCell next;

   public LLCell(object v, LLCell n)  {
     value = v;
     next = n;
   }
}
```

Implementing a stack with a linked list



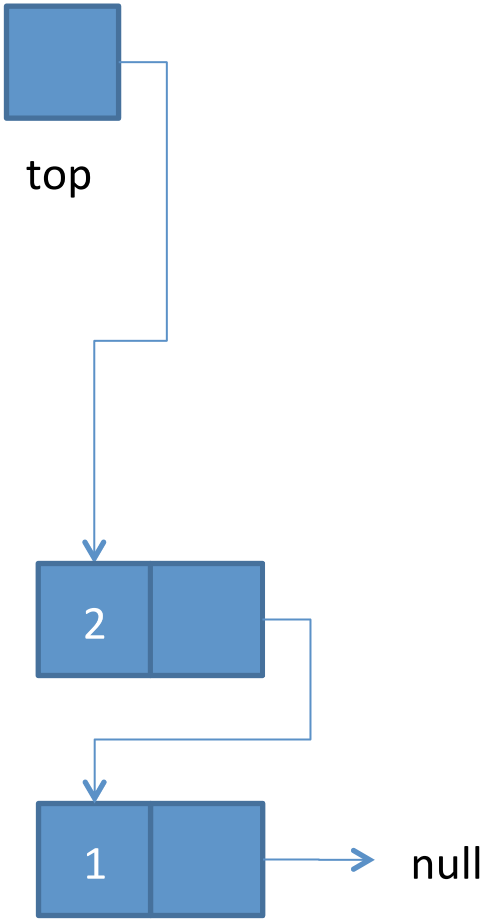
s = new LLStack();

Implementing a stack with a linked list



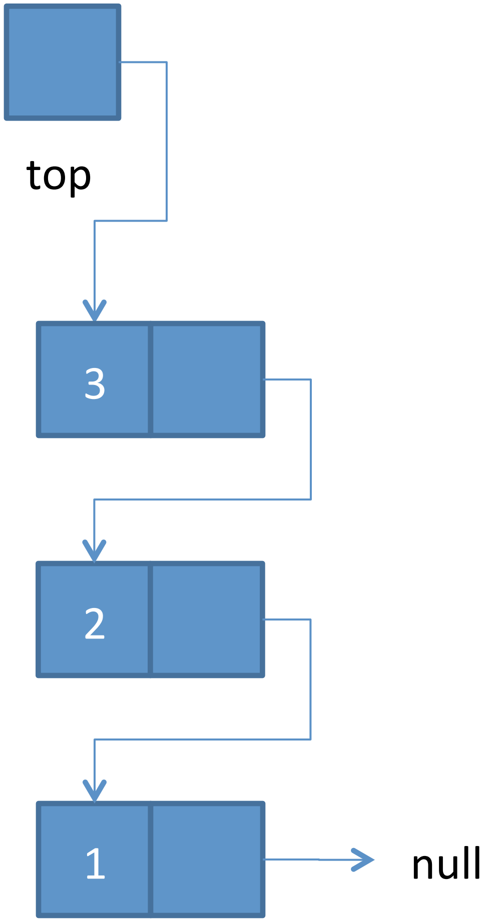
```
s = new LLStack();
s.Push(1);
```

Implementing a stack with a linked list



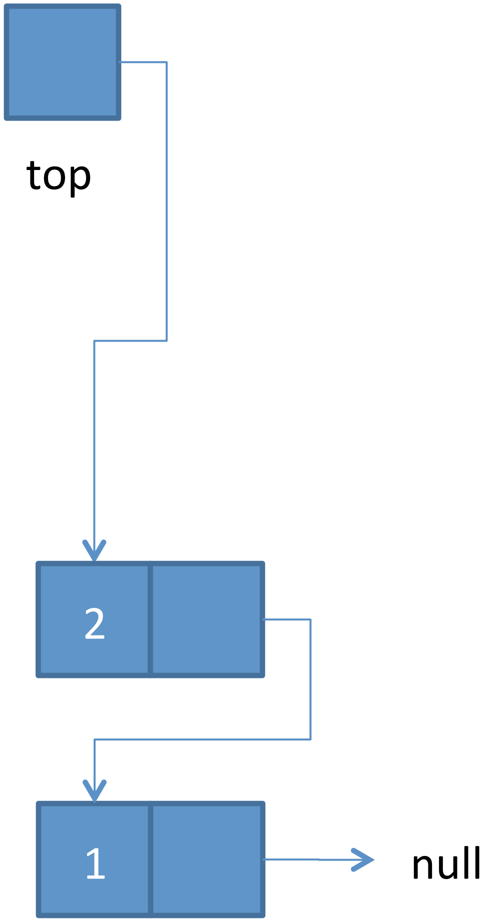
```
s = new LLStack();
s.Push(1);
s.Push(2);
```

Implementing a stack with a linked list



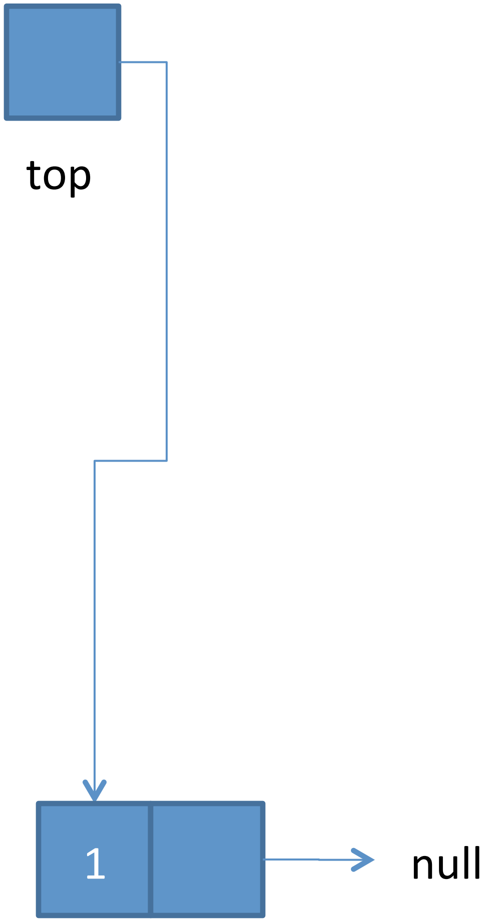
```
s = new LLStack();
s.Push(1);
s.Push(2);
s.Push(3);
```

Implementing a stack with a linked list



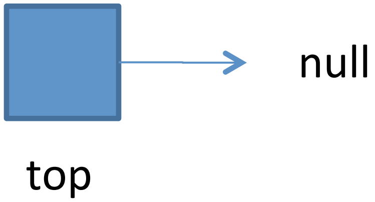
```
s = new LLStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop();        // returns 3
```

Implementing a stack with a linked list



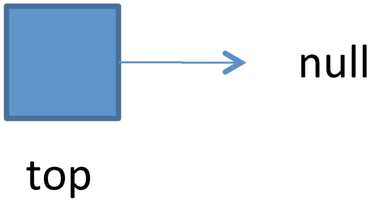
```
s = new LLStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop();      // returns 3
s.Pop();      // returns 2
```

Implementing a stack with a linked list



```
s = new LLStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop();        // returns 3
s.Pop();        // returns 2
s.Pop();        // returns 1
```

Implementing a stack with a linked list



```
s = new LLStack();
s.Push(1);
s.Push(2);
s.Push(3);
s.Pop();       // returns 3
s.Pop();       // returns 2
s.Pop();       // returns 1
s.IsEmpty      // true
```

Queues

- From the French for "**line**" or "tail"
- A specialized type of sequence where
  - **Additions** can only be performed at the "**end**" or "**tail**"
  - **Removals** can only be performed at the "**beginning**" or "**head**"

flickr user DaveKav

Queues

Like stacks, queues give their add and delete operations funny names

- **Enqueue**(item)
  Adds item to the end of the queue
- **Dequeue**()
  Removes item at the head of the queue and returns it

flickr user DaveKav

Queues

Like stacks, queues can be implemented using any data structure for representing sequences

- Arrays
- Linked lists

flickr user DaveKav

Simple queue implementation using a static array

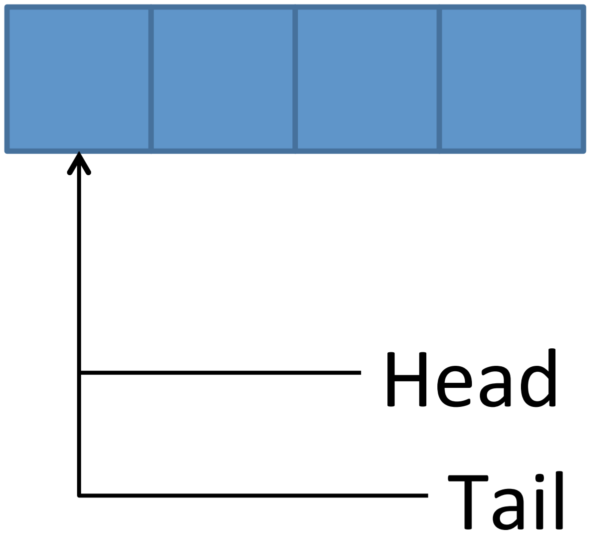
- One simple way to implement a queue is to use a **fixed-size array**
  - Limits number of elements that can be in the queue

- Also uses two fields to keep track of data in the queue
  - Head: index in the array of the next element to dequeue
  - Tail: index in the array of where the next enqueued element should be stored

```
object[] data = new object[100];
int head;
int tail;

void Enqueue(object o) {
   data[tail] = o;
   tail = (tail+1)%data.Length;;
}

object Dequeue() {
   object item = data[head];
   head = (head+1)%data.Length;
   return item;
}
```

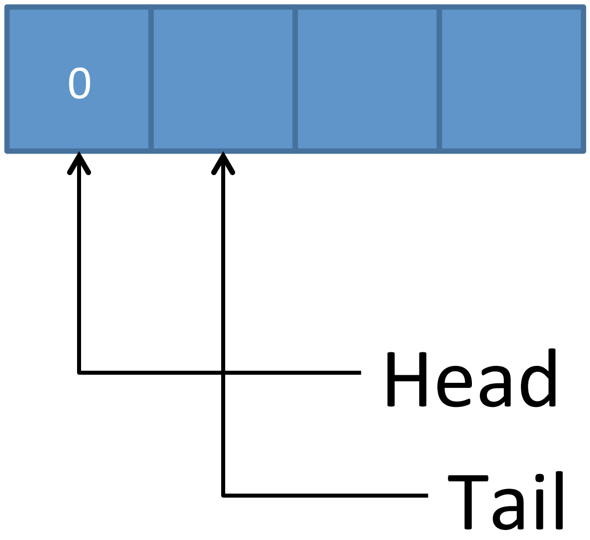
Array-based queue

(with a capacity of 3 elements)



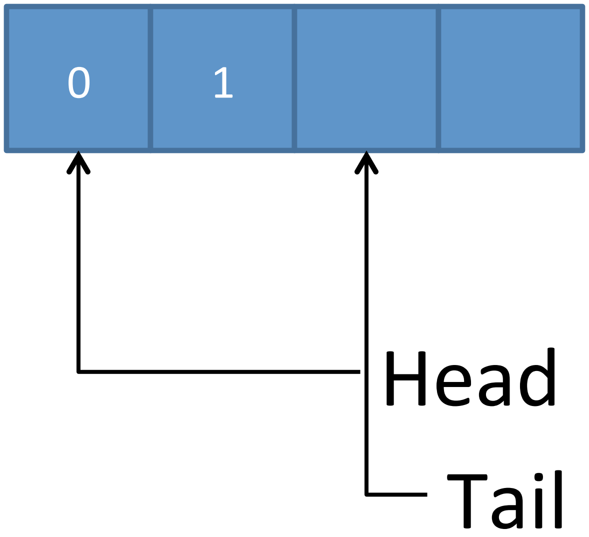
Array-based queue

- Enqueue(0)



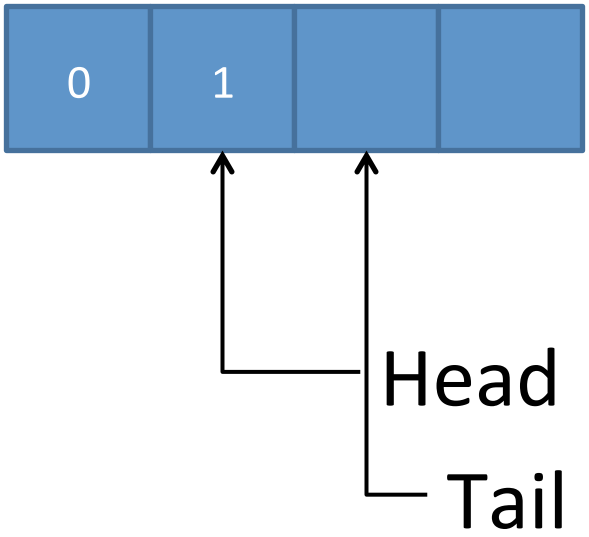
Array-based queue

- Enqueue(0)
- Enqueue(1)



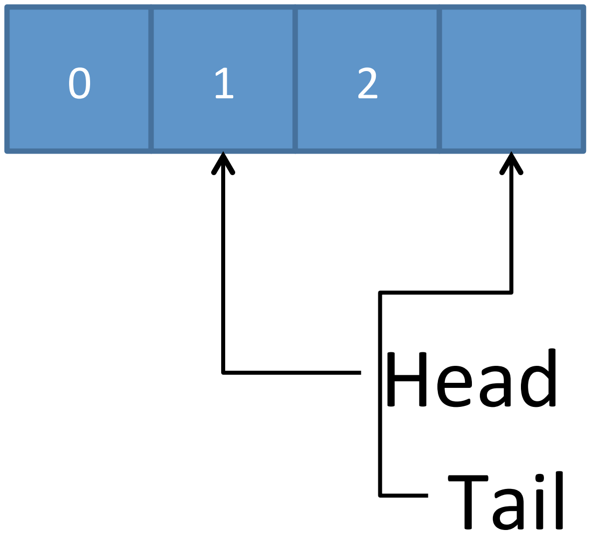
Array-based queue

- Enqueue(0)
- Enqueue(1)
- Dequeue()
  - Returns 0



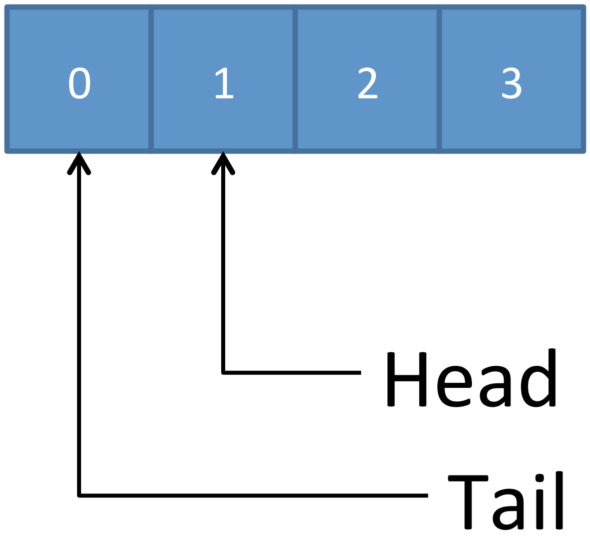
Array-based queue

- Enqueue(0)
- Enqueue(1)
- Dequeue()
    - Returns 0
- Enqueue(2)



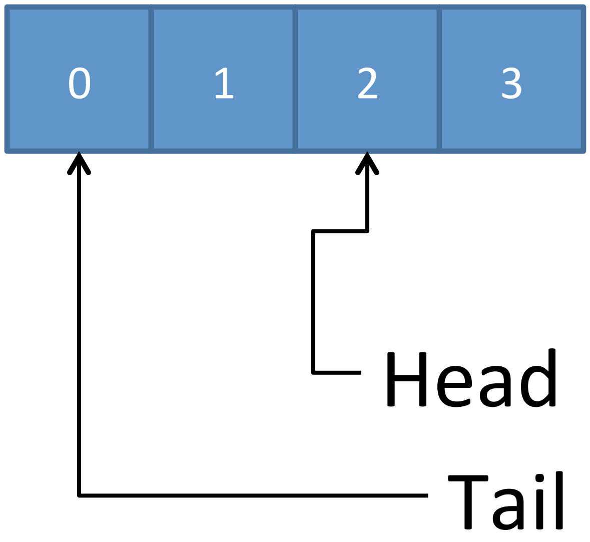
Array-based queue

- Enqueue(0)
- Enqueue(1)
- Dequeue()
  - Returns 0
- Enqueue(2)
- Enqueue(3)



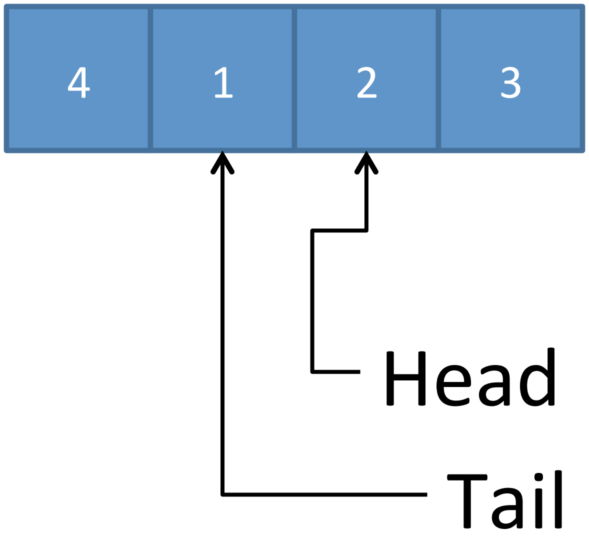
Array-based queue

- Enqueue(0)
- Enqueue(1)
- Dequeue()
  - Returns 0
- Enqueue(2)
- Enqueue(3)
- Dequeue()
  - Returns 1



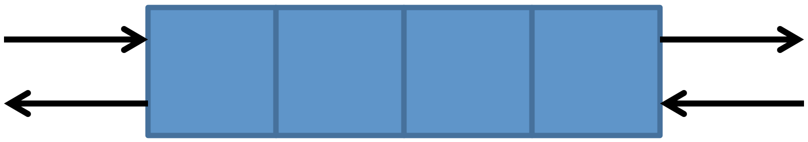
Array-based queue

- Enqueue(0)
- Enqueue(1)
- Dequeue()
  - Returns 0
- Enqueue(2)
- Enqueue(3)
- Dequeue()
  - Returns 1
- Enqueue(4)



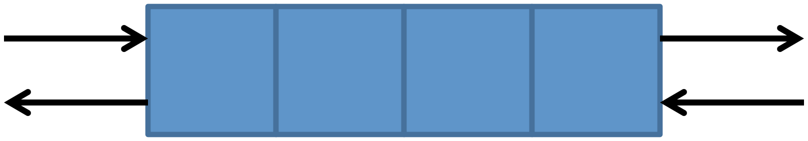
Deques

- Pronounced “**deck**”
- **Double-ended queue**
- Specialized sequence in which **additions and deletions**
  - Can be made on **either side**
  - But only on the sides



Deques

- Deques generalize stacks and queues



- Behave **like a stack**
  - If you only add/remove from one side

- Behave **like a queue**
  - If you add from one side
  - And remove from the other

Reading

- CLRS, Chapter 10 (**Elementary Data Structures**)
  - Sections 1-3
    - 11.1 Stacks and queues
    - 11.2 Linked lists
    - 11.3 Implementing pointers and objects

Assignment 1

- **Implement queues**
  - Using arrays
  - And linked lists
- Do a **fuller implementation** than discussed in class
  - Extra methods, like IsEmpty
  - Should check for error conditions and throw appropriate exceptions
- **Implement deques** using doubly-linked lists

- Test using **automated testing tools** in Visual Studio
  - We provide a full set of test cases
  - **Worry-free**: if code passes tests, you'll get full credit
  - Requires VS 2013 Ultimate,
    - Will not work with C# Express
- Out: Monday, April 13
  - Due Friday, April 24
  - **Do not wait until the last minute**