INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 3

Instructor: James Pope

Substitute: Kevin Andrea

Email: jpope8@gmu.edu

Today

- Last Class
  - Big-O, Arrays, Searching (Linear), Dynamic Lists, Sorting (Insertion)
- Today
  - Linked Lists
  - Queues
  - Stacks
  - Recursion
  - PA2/3

Questions?

Last Class: Big-O

- Finish this definition:
  - T(n) is O(F(n)) if...

Last Class: Big-O

- Finish this definition:
  - T(n) is O(F(n)) if there are... positive constants **c** and **n0** …such that...
    - when n is... greater than or equal to n0
    - T(n) is...less than or equal to c F(n)

Big-O: $3n^2+n+30$ is $O(n^2)$

n	$3n^2 + n + 30$	n^2	$c=4 \rightarrow 4n^2$
1	$3(1^2)+1+30 = 34$	$1^2 = 1$	4
2	$3(2^2)+2+30 = 44$	$2^2 = 4$	16
3	$3(3^2)+3+30 = 60$	$3^2 = 9$	36
4	$3(4^2)+4+30 = 82$	$4^2 = 16$	64
5	$3(5^2)+5+30 = 110$	$5^2 = 25$	100
6	$3(6^2)+6+30 = 144$	$6^2 = 36$	144
7	$3(7^2)+7+30 = 184$	$7^2 = 49$	196
8	$3(8^2)+8+30 = 230$	$8^2 = 64$	256
9	$3(9^2)+9+30 = 282$	$9^2 = 81$	324
10	$3(10^2)+10+30 = 340$	$10^2 = 100$	400

What is n_0?

Big-O Practice

- Do big-O and order these functions:
  - 5 n log n + 74
  - 33 log n + n^2
  - 2 sqrt(n)
  - log log n + n!
  - n^2 + 2^n
  - 6 n log n – n
  - T(n) = n^4 if n < 100, else T(n) = n^3

Questions?

Selection Sort (Big-O)

- Sorting an array of numbers:
  - Find the smallest number: O(__)
  - Put it first: O(__)
  - Find the second smallest... put it next
  - How many times? So O(__)
- What is Big-O for:

```
for(int i = 0; i < n; i++)
  for(int j = i+1; j < n; j++)
    // find smallest, swap
```

Sums...

- Arithmetic Series (sum of a sequence)

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- O(__)?

Big-O of Dynamic Arrays

- Dynamic Arrays (ArrayList)
  - Get/Set is O(__)
  - Add/Remove from end is O(__)
    - Amortized add: O(__)
  - Add/Remove from beginning is O(__)
  - Add/Remove from middle is O(__)
  - Finding things is O(__)

Insertion Sort

- Start at the beginning

  ...and when you get to the end stop

- Demo

- Resource with animations if you forget this:
  - http://en.wikipedia.org/wiki/Insertion_sort

Properties of Insertion Sort

- Worse Case?
- Best Case?
- In-place
  - only O(1) additional memory space
    - remember O(1) does not always equal 1
- Stable
  - relative order of equal elements preserved

Last Class: Insertion Sort

```
//assume: int[] ints
for(int i = 0; i < ints.length; i++)
{
  int j = i;
  while(j > 0 && ints[j] < ints[j-1])
  {
    //swap ints[j] and ints[j-1]
    int temp   = ints[j];
    ints[j]    = ints[j-1];
    ints[j-1] = temp;
    j--;
  }
}
```

Alternate: Insertion Sort

```
//assume: int[] ints
for(int i = 0; i < ints.length; i++)
{
  for(int j = i; j > 0; j--)
  {
    if(ints[j] < ints[j-1])
    {
      //swap ints[j] and ints[j-1]
      int temp   = ints[j];
      ints[j]    = ints[j-1];
      ints[j-1] = temp;
    }
    else break;
  }
}
```

Questions?

Linked Lists

- Data structure along the lines of a treasure hunt
  - Start at the beginning
  - At each “stop” find an item and the clue to the next place to look

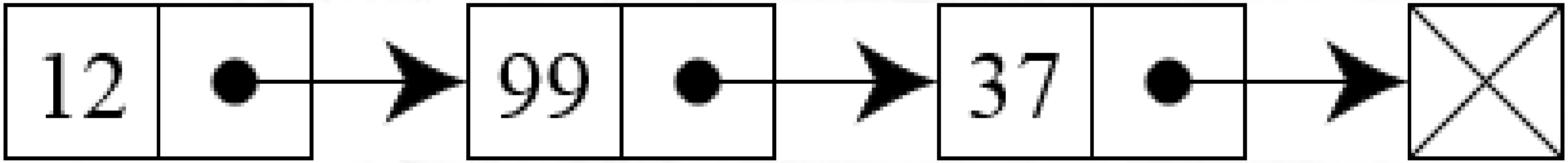


Image Source: http://en.wikipedia.org/wiki/File:Singly-linked-list.svg

Node Data Structure

- What fields do we need to store a node?
  - What are the defaults?
  - Any advantage keeping reference to last node?
- What methods do we need?
  - What methods should we have?

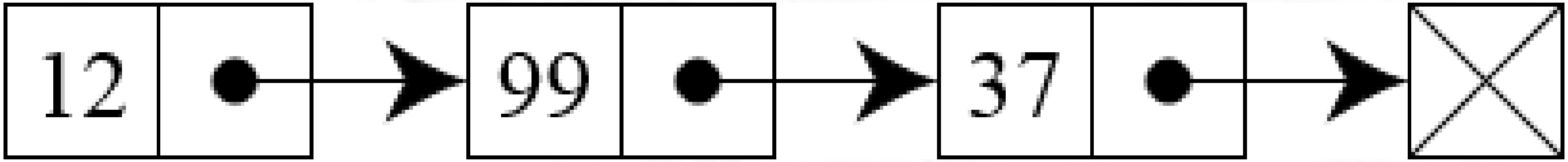


Image Source: http://en.wikipedia.org/wiki/File:Singly-linked-list.svg

List Data Structure

- What fields do we need?
  - What are the defaults?
- What methods do we need?
  - This is a replacement for an array, so...

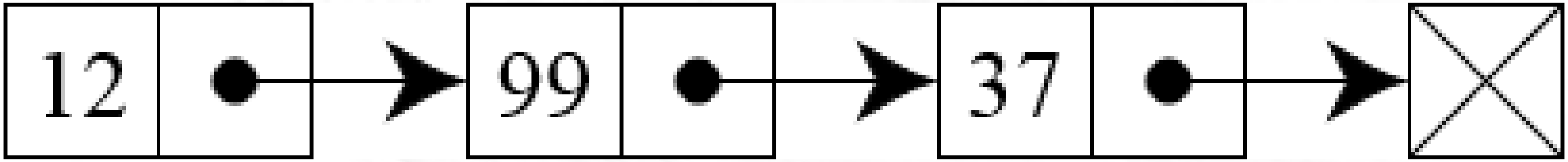


Image Source: http://en.wikipedia.org/wiki/File:Singly-linked-list.svg

Last Class - List Operations

```
public interface List<Type>
{
  public Type get(int i);

  // overwrites
  public void set(int i, Type item);

  // add to the end
  public void add(Type item);

  // remove item at index i from list
  public Type remove(int i);

  // insert to front or middle
  public void insert(int i, Type item);

  // sequential search to find index, -1 not found
  public int search(Object o); // aka find
}
```

Linked List Array Emulation

- Minimum Required to Emulate an Array
  - Get / Set element
  - Add
  - Get size
- Cool new things array cannot inherently do
  - Remove
  - Insert
  - Add beyond current capacity, can't (usually) run out of space (just like a dynamic array)

How do you...

- How do you do things like: get the item before 37
  - Can we make a work around?
- How do you walk “backwards” through the list?
  - Can we modify things to help?

Linked List Variants

- Node Fields
  - Reference to next node (“singly”)
  - Reference to previous and next node (“doubly”)
- List Fields
  - Keep reference to front node (“single-ended”)
  - Keep reference to front and last node (“double-ended”)
- Easy to keep/maintain reference to front and last, unless specified otherwise, will generally assume double-ended
  - Singly Linked List (“singly” nodes)
  - Doubly Linked List (“doubly” nodes)

Linked List

Big-O ?

Operation Implementation	get set	add remove last	insert remove front	insert remove middle	search
Singly Linked List					
Doubly Linked List					

Linked List
Big-O

Operation Implementation	get set	add remove last	insert remove front	insert remove middle	search
Singly Linked List	N	1 or N*	1	N	N
Doubly Linked List	N	1	1	N	N

* Add is 1 for double-ended, remove last is still N

- Only use memory proportional to N
- “Locality of reference” poor compared to array based

Questions?

Arrays, Dynamic Arrays, & Linked Lists

- Array / Static Array - “row” of memory
    - can run out of space
- Dynamic Arrays - arrays that can grow
    - cost to copy repeatedly (not so bad)
    - Insert/remove expensive (not good at all)
- Linked Lists - tiny blocks of memory “linked” together
    - no “quick” memory access
    - extra memory to represent compared to array

Arrays vs. Other

- Arrays are simple
  - get/set anything
  - add/remove is obvious (need size variable)
  - very clear how data is laid out
- Just about every other data structure is less so
  - get/set nontrivial
  - must preserve some internal structure - control access
  - element-by-element access takes work (time)

List Operations Summary

Big-O

Operation Implementation	get set	add remove end	insert remove begin	insert remove middle	search	Grow Shrink
Array	1	-	-	-	-	No
Static Array	1	1	N	N	N	No
Dynamic Array	1	1*	N	N	N	Yes
Single Linked List	N	N	1	N**	N	Yes
Doubly Linked List	N	1	1	N**	N	Yes
???	1	1	1	1???	N	Yes

* Amortized analysis
** Have to search first N, then insert is constant

- Though arrays are limited in functionality, constants for arrays are much faster

Questions?

Iterator

- Give access to all the items in a collection in some unspecified order
  - Move around next()
  - Check for next hasNext()
- Sometimes: previous(), hasPrevious(), add(), remove()
- Conceptually the iterator has a position between two elements
- Client unaware of how items are stored internally and typically doesn't need to know

Iterator Operations

From Javadocs 1.7

http://www.eecs.yorku.ca/course_archive/2011-12/W/2011/lectures/04%20The%20Java%20Collections%20Framework.pdf

```
/**
 * Returns true if the iteration has more elements
 * @return true if more, false otherwise
 */
public boolean hasNext();

/**
 * Returns the current element and moves (if possible)
 * to next element making it the current element.
 * @return next element in the iteration
 * @throws NoSuchElementException if no more elements
 */
public Type next();

/**
 * Removes the last element returned by next.  This method can be called
 * only once per call to next.  The behavior of an iterator
 * is unspecified if the underlying collection is modified while iterating.
 * @throws UnsupportedOperationException if not supported
 * @throws IllegalStateException if the next method has not
 *         yet been called, or remove method has already
 *         been called after the last call to next
 */
public void remove();
```

Types of Iterators

Iterator

Removing

```
LL l = new LL([A, B, C, D])
itr = l.iterator()

itr.next()   [A, B, C, D]
              ^
itr.next()   [A, B, C, D]
                 ^
itr.remove() [A, C, D]

itr.next()   [A, C, D]
                 ^
itr.remove() [A, D]

itr.remove() [A, D] //Error
```

ListIterator extends Iterator

Next/Previous

```
LL l = new LL([A, B, C, D])
itr = l.iterator()

itr.next()     [A, B, C, D]
                ^
itr.next()     [A, B, C, D]
                   ^
itr.previous() [A, B, C, D]
                   ^
itr.previous() [A, B, C, D]
                ^
itr.next()     [A, B, C, D]
                ^
itr.remove()   [B, C, D]
```

Implementing Iterator

- The next() and hasNext() are straight forward to properly implement
- The remove() operation is difficult
  - Keep index of previous index
  - Check for two sequential remove() calls
  - Address concurrent modification (multiple iterators)
- Implementing an Iterator is easy if the remove() is omitted. Typically this is done.
  - Throw UnsupportedOperationException
  - Document for clients

Other Thoughts...

- Where does it point when it's created?
- For add/remove, where are they added?
  - remember conceptually the iterator is...
- Can you have multiple iterators?
- What methods might be difficult for a singly linked list (previous())?

ConcurrentModificationException

```
it1 = list.iterator();
it2 = list.iterator();
it1.remove();
it2.next(); // Error
```

- Doesn't try to coordinate multiple iterators changing. Note this is different issue than multiple thread access (will not cover).
- Easy for reading/viewing
- Difficult for modification
- A generally recurring pattern in CS:
  - multiple simultaneous actors

Iterator that checks Concurrent Modification: remove(int)

```
private class Itr implements Iterator<Type>
{
  int expectedModCount = modCount; //modCount in outer
  ...
  public Type next()
  {
    if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
    ...
  }

  public Type remove(int index)
  {
    if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
    remove(); // removes from the list
    expectedModCount = ++modCount;
    ...
  }
}
```

Will generally avoid and not support Iterator.remove

Java: Iterable<Type>

- An Iterable is an object that can be iterated, i.e. returns an Iterator

```
/**
 * Object that can be iterated
 */
public interface Iterable<Type>
{
  /**
   * Returns an iterator over elements of type
   * @return itr
   */
  public Iterator<Type> iterator();
}
```

Anonymous Class Style

```
public class MyClass implements Iterable<Type>
{
  public Iterator<Type> iterator()
  {
    return new Iterator<Type>()
    {
      /* Iterator code here */
    };
  }
}
```

Inner Class Style

```
public class MyClass implements Iterable<Type>
{
  ...
  private class Itr implements Iterator<Type>
  {
    /* Iterator code here */
  }

  public Iterator<Type> iterator()
  {
    return new Itr<Type>();
  }
}
```

List Interface updated to be Iterable

```
public interface List<Type> extends Iterable<Type>
{
  ...
  public Iterator<Type> iterator();
  ...
}
```

Iterable Dynamic Array

```
public Iterator<Type> iterator()
{
    // Create anonymous Iterator.
    // Note: Creates class and instance of it
    return new Iterator<Type>()
    {
        private int current = 0;

        public boolean hasNext()
        {
            return current < size;
        }

        public Type next()
        {
            if( current >= size )
            {
                throw new NoSuchElementException("No more elements");
            }
            return get(current++);
        }

        public void remove()
        {
            throw new UnsupportedOperationException("remove");
        }
    };
}
```

Enhanced for-each

```
DynamicArray<String> list = new DynamicArray<String>();
list.add("Alpha");
list.add("Bravo");
list.add("Charlie");
list.add("Delta");

// Example iterator-while loop, little verbose
Iterator<String> iter = list.iterator();
while( iter.hasNext() )
{
    String item = iter.next();
    Stdio.println(item);
}

// Enhanced for-each shortcut, compiler converts to
// the above equivalent iterator-while loop
// Can only use if the object is Iterable
for (String item : list)
{
    Stdio.println(item);
}
```

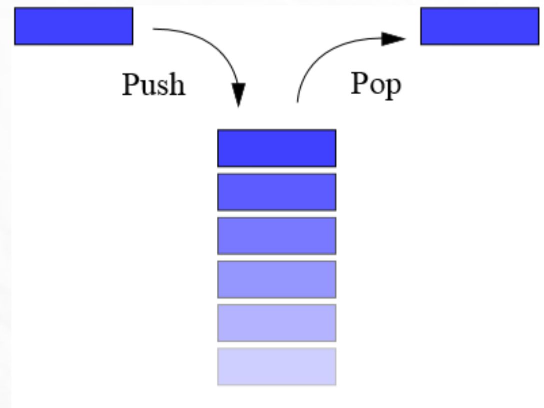
Why do we need Iterable/Iterator?

- Common operation is to traverse all elements in the data structure. Each data structure is fundamentally different.
- Without Iterable/Iterator, clients would have to develop code tailored to each data structure in order to accomplish this.
- Using common interface, want to traverse...
  - Lists (Dynamic Arrays, Linked Lists, ...)
  - Stacks and Queues
  - Trees (Iterators for different traversals)
  - etc.

Questions?

Stacks & Queues

- Stack
  - Data structure that works like a... stack (e.g. a stack of paper)



- Queue
  - Data structure that works like a... queue (or a “line” if you aren't British)

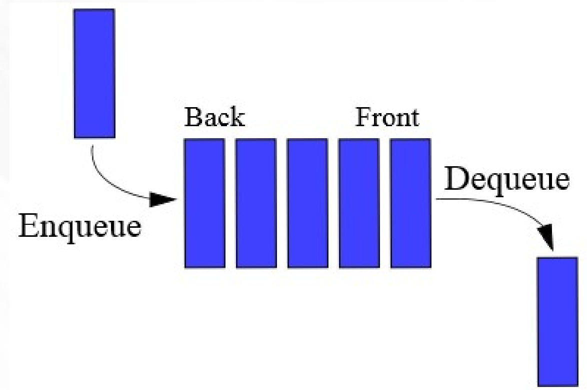


Image Source: http://en.wikipedia.org/wiki/File:Data_stack.svg and http://en.wikipedia.org/wiki/File:Data_Queue.svg

Stacks

- Like a stack of paper
  - the first paper on the stack is on the bottom, usually you pick them up from the top
  - also called a LIFO queue, last-in-first-out

Stack Operations

```
public interface Stack<Type>
{
    // adds item to top of stack
    public void push(Type item);

    // removes and returns item from top
    public Type pop();

    // return, but not remove, top item
    public Type peek();

    // true if no items, false otherwise
    public boolean isEmpty();

    // number of items in the stack
    public int size();
}
```

White Board Stacks

- Dynamic Array (or Array w/size)
  - Typical solution, good constants.
- Linked List
  - Can use single linked list.
  - Larger constant, but consistent performance

Queues

- Like lines to buy things
  - first person in the line gets to buy things first
  - technically the above is a FIFO queue, first-in-first-out

Queue Operations

```
public interface Queue<Type>
{
    // adds item to end of queue
    public void enqueue(Type item);

    // removes item from front of queue
    public Type dequeue();

    // return, but not remove, front item
    public Type peek();

    // true if no items, false otherwise
    public boolean isEmpty();

    // number of items in the queue
    public int size();
}
```

Queues - White Board

- Array w/size (or DynamicArray)
  - Typically called “Circular Queue”. Can be dynamically sized. Either way, more complicated to program than using the linked list approach.
- Linked List
  - Relatively easy implementation using singly nodes with first/last node reference.
  - Can also use Doubly Linked List but unnecessary and requires more memory for the extra previous reference per node

Queues – Singly/Double-Ended

- enqueue(Type item)
  - Create new node with the item
  - If empty, front = new node
  - Else last.next = new node
  - Update last = new node
- Type dequeue()
  - Node temp = front node
  - Update front = front.next
  - If empty, last = null (no loitering)
  - Return temp.item

Why the restrictions?

e.g. no insert(), set(), remove()

- Simple data structures
  - focus on limited operations
  - can be made out of arrays
- Good for representing time-related data
  - call stack
  - packet queues
- Why? Because good worst cases
  - O(1) for all supported operations
  - O(n) space

Stack

Big-O

Operation Implementation	push	pop	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

Easiest to implement Dynamic Array, constants are better

Queue

Big-O

Operation Implementation	enqueue	dequeue	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

For Queue, Dynamic Array is called a "Circular Queue"
Easiest to implement Singly (double-ended) Linked List

O(1)??

- What is big-O to add or remove items from the front of an array?
  - How did we get O(1) for enqueue/dequeue
  - Circular queues
- The operations are very similar. API could be the same, meaning is different, for historic reasons, best to use specific terms.
- Can you implement a Queue with Stack(s)? How might this work? See Queue2Stacks

Questions?

Priority Queues

- Highest priority (min/max) comes out first
  - Not FIFO or LIFO
- Nodes have another component
  - usually called “key” (aka “the thing you look for and/or sort by in a set of data”)
- How can we implement this with arrays or linked lists?
- Used directly in applications and supports many other algorithms, e.g. graph processing

(Max) Priority Queue API

Derived from Weiss 6.9

```
public interface MaxPQ
    extends Iterable<Comparable>
{
    public void insert(Comparable key);
    public Comparable delMax();
    public Comparable findMax();
    public int size();
    public boolean isEmpty();
}

// MinPQ replaces delMax/findMax
// with delMin/findMin
```

Simple Priority Queue Approaches

- Unordered array
  - insert?
  - delMax?
  - findMax?
- Ordered array (See OrderedArrayMaxPQ.java), can use the InsertionSort approach to order inserted items
  - insert?
  - delMax?
  - findMax?

Priority Queue
Big-O

Operation Implementation	insert	delMax	findMax
Unordered Array	1	N	N
Ordered Array	N	1	1
???	lg(N)	lg(N)?	1

Set of Priority Queues

- Single queue
  - insert item at desired location (like inner loop of insertion sort)
- Set of queues
  - One queue per “key”
    - e.g. “high” and “low” priority items
  - dequeue()
    - if “high” not empty, dequeue from “high”
    - else dequeue from “low”

Questions?

Recursion

- Call a function/method from inside the same function/method
- Idea: keep doing the same thing, reducing the problem
  - smaller subset of the problem
  - one step closer to the answer
  - subsets should not overlap
- Key components
  - recursive case (when to keep going)
  - base case (when to stop)

Recursion Example 1

```
int sumValues(int start, int end)
{
  int sum = 0;
  for(int i = start; i <= end; i++)
    sum += i;
  return sum;
}

int sumValues(int start, int end)
{
  if(start == end)
    return end;
  return start + sumValues(start+1, end);
}
```

Recursion Example 2

Derived: http://users.dickinson.edu/~braught/courses/cs132f01/classes/code/Factorial.src.html
Weiss 7.3

```
int factorial(int n)
{
  // Base Case
  if(n <= 1) return 1;
  // Recursive Case
  else return n * factorial(n-1);
}

int x = factorial(5);
```

Recursion Example 2

- If the method involves inspecting the recursive call's value before returning, then it must be placed on the call stack (activation record)

```
Top -> AR5: n=1 return=1                  Unwinds
       AR4: n=2 return=2*factorial(1)
       AR3: n=3 return=3*factorial(2)
       AR2: n=4 return=4*factorial(3)
       AR1: n=5 return=5*factorial(4)
```

- Note: Stack memory consumed, how much?

Questions?

Binary Search

- Review: Linear Search
- We learned sorting...

  ... use it!
- Common technique: reduce search space
- Whiteboard...

- Recursive method

Binary Search

- Requires?
- Method?
- Big-O?
    - worst case?
    - best case?

Binary Search

- Requires: sorted list
- Method: see next slide, or:
  - http://en.wikipedia.org/wiki/Binary_search_algorithm
- Big-O
  - worst case: O(lg n)
    - why?
  - best case: O(1)
    - why?

Binary Search

```
// Note: not real code...assumes list is sorted!!!
int binarySearch(list, value)
{
  if(list is empty)
    Return -1
  if(middle of list == value)
    return middle index
  if(middle of list > value)
    return binarySearch(first half of list, value)
  if(middle of list < value)
    return binarySearch(second half of list, value)
}
```

Questions?

Assignments: PA1 / PA2 / PA3

- PA1: Grades posted
- PA2: Due Tomorrow
- PA3
  - Make an ADT (abstract data type)
  - Implement the QueueAPI Interface
  - Use anonymous/inner classes
  - Implement Iterable/Iterator

Let's Look at the Files

Free Question Time!