EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

Linear Data Structures

Relevant Sections

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)

Note: Some slides in this presentation have been adapted from the authors' slides.

Arrays and Java

- the length of an array named a can be accessed using $a.length$ (fixed at the time of construction).

```java
int[] A = new int[10];
```

```java
int[] A={12, 24, 37, 53, 67};
for (int i=0; i < A.length; i++) {..}
```

- the cells of an array, a, are numbered $0, 1, 2, \ldots$ up through $a.length - 1$, and the cell with index k can be accessed with syntax $a[k]$ in $O(1)$ time
- array elements are placed contiguously in memory

Declaring Arrays

- assignment to a literal form when initially declaring the array, using a syntax as:

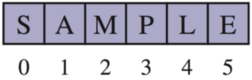
```
elementType[] arrayName = {a,b,c,d,e}
```

- elementType: any Java base type or class name, arrayName: any valid Java identifier. Initial values: the same type as the array.
- The second way to create an array is to use the new operator. However, because an array is not an instance of a class, we use the syntax:

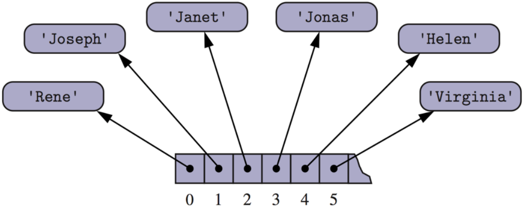
```
new elementType[length]
```

Arrays of Characters or Object References

- An array can store primitive elements, such as characters



- An array can also store references to objects.



More on Java Arrays

- Java arrays are homogeneous
  1. all array components must be of the same (object or primitive) type
  2. an array of an object type can contain objects of any respective subtype
- Memory management
  1. allocated dynamically by means of new
  2. automatically deallocated when no longer accesses

Automatically initialized with

1. 0, for an array of int[] or double[] type
2. false, for a boolean[] array
3. null, for an array of objects

Common Errors

- Unallocated arrays
  int[] numbers;
  numbers[2] = 100;

- ArrayIndexOutOfBoundsException: thrown at an attempt to index into array A using a number larger than $A.length - 1$
  helps Java avoid ‘buffer overflow attacks’

Insertion and Removal

Insertion

```java
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
  int newScore = e.getScore();
  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
    if (numEntries < board.length)          // no score drops from the board
      numEntries++;                          // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries - 1;
    while (j > 0 && board[j-1].getScore() < newScore) {
      board[j] = board[j-1];                 // shift entry from j-1 to j
      j--;                                   // and decrement j
    }
    board[j] = e;                            // when done, add new entry
  }
}
```

Insertion and Removal - 2

Removal

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                       // save the object to be removed
  for (int j = i; j < numEntries − 1; j++)         // count up from i (not down)
    board[j] = board[j+1];                         // move one cell to the left
  board[numEntries −1 ] = null;                    // null out the old last score
  numEntries−−;
  return temp;                                     // return the removed object
}
```

Multidimensional Arrays - Examples

```
public class UseArrays{
    public static void main(String[] args) {
        int[][] nums = new int[2][3];
        nums[0][0] = 1;

        int[][] nums2;
        nums2 = new int[2][];
        for (int i=0; i<2; i++) {
            nums2[i] = new int[3];
        }
        nums2[0][0] = 1;

        int[][] nums3 = {{1,2,4},{3,4,5}};
        nums3[0][0] = 1;
    }
}
```

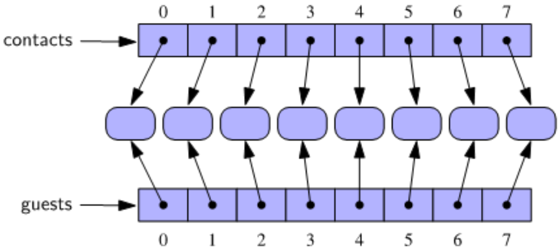
Copying vs Cloning

```
public class UseArrays{
    public static void main(String[] args) {
        int[] A={12, 24, 37, 53, 67};

        int[] B = A;
        System.out.println("A[2]="+A[2]+",B[2]="+B[2]);
        /* A[2]=37, B[2]=37 */
        B[2] = 5;
        System.out.println("A[2]="+A[2]+",B[2]="+B[2]);
        /* A[2]=5, B[2]=5 */
        int[] C={12, 24, 37, 53, 67};
        int[] D = C.clone();
        D[2] = 5;
        System.out.println("C[2]="+C[2]+",D[2]="+D[2]);
        /* C[2]=37, D[2]=5 */
    }
}
```

Deep Copying vs Shallow Copying

- Shallow copy of an object – exact copy of all the fields of original object.
- Any references to other objects as fields are copied into clone object, copy of those objects are not created.
- Clone does a shallow copy



From James Elder's slides

Java.util.Arrays API

- equals(A,B): returns true if A and B have an equal number of elements and every corresponding pair of elements in the two arrays are equal
- fill(A,x): store element x into every cell of array A
- sort(A): sort the array A in the natural ordering of its elements
- binarySearch([int] A, int key): search the specified array of sorted ints for the specified value using the binary search algorithm (A must be sorted)

Equality Testing of 2d Arrays

Methods: ==, Arrays.equals, Arrays.deepEquals

```
int[][] nums = new int[2][3];
nums[0][0] = 1;

int[][] nums2;
nums2 = new int[2][];
for (int i=0; i<2; i++) {
    nums2[i] = new int[3];
}
nums2[0][0] = 1;

int[][] nums4 = new int[2][3];
nums4[0][0] = 1;
```

From Arrays to Arraylists

- array size cannot be changed
- Insertion / deletion in an array is expensive because elements have to be moved
- Arraylists (Ch 7): adaptively grow and shrink an array

Ch 7.1: The List ADT

The java.util.List interface; methods:

- size()
- isEmpty()
- get(i)
- set(i,e)
- add(i,e)
- remove(i)

List Operations

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

The Array List ADT

- Like an array, the Array List ADT stores a sequence of arbitrary objects
- An element can be accessed, modified, inserted or removed by specifying its rank (number of elements preceding it)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

Arraylist API

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

- size(), isEmpty(), add, addAll, clear(), clone(), contains(Object o), get(int index), int indexOf(Object o), remove(int index), removeAll)
- ensureCapacity(int minCapacity): Increases the capacity of this ArrayList instance, if necessary
- `Iterator<E> iterator()`
  Returns an iterator over the elements in this list in proper sequence.
- toArray()

Java Reminder: Generics

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

- for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts.
- allows us to define a class in terms of a set of formal type parameters
- the formal type parameters are later specified when using the generic class as a type elsewhere in a program.

Java Reminder: Syntax for Generics

- Types can be declared using generic names

```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {                    // constructor
    first = a;
    second = b;
  }
  public A getFirst( ) { return first; }
  public B getSecond( ) { return second;}
}
```

- instantiated using actual types:

```
Pair<String,Double> bid;
```

Arraylist: Implement using Arrays

- Arrays: obvious choice for implementing the list ADT
- $A[i]$ stores (a reference to) the element with index i
- get(i) and set(i, e) methods are easy to implement by accessing $A[i]$ (if i is a valid index)
- Insertion, removal need shifting $\Omega(n)$ elements in the worst case

Performance

- The space used by the data structure is $O(n)$
- Indexing the element at i takes $O(1)$ time
- add and remove run in $O(n)$ time
- In an add operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one ...

Growing Arrays - Choices

How large should the new array be?

- Incremental strategy: increase the size by a constant c
- Doubling strategy: double the size

Question: How do we evaluate which is better?

Problem: Each doubling has different cost.

Amortized Analysis

Key Idea: compare the strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations

- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant
- The total time T(n) of a series of n push operations is proportional to

$$\begin{aligned} T(n) &= n + c + 2c + 3c + 4c + \ldots + kc \\ &= n + c(1 + 2 + 3 + \ldots + k) \\ &= n + ck(k+1)/2 \end{aligned}$$

- Since c is a constant, $T(n) \in \theta(n + k^2) = \theta(n^2)$
- Thus, the amortized time of a push operation is $\theta(n)$

Double Strategy Analysis

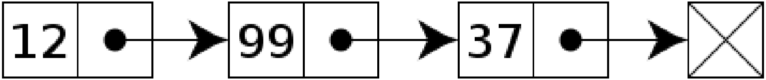
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$\begin{aligned} T(n) &= n + 1 + 2 + 4 + 8 + \ldots + 2^k \\ &= n + 2^{k+1} - 1 \\ &= 3n - 1 \\ T(n) &\in \theta(n) \end{aligned}$$

- The amortized time of a push operation is $\theta(1)$

The Linked List Data Structure

- A singly linked list is a data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores an element and a link to the next node



By Lasindi - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2245162

Linked List Class

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;            // reference to the element stored at this node
    private Node<E> next;         // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

Linked List Class - 2

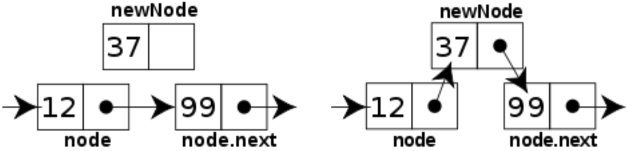
```
public class SinglyLinkedList<E> {
...  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {               // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

Java Reminder: Nested Classes

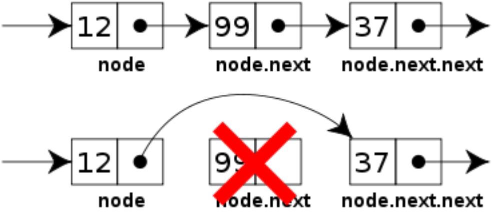
- a class definition can be nested inside the definition of another class.
- main use: defining a class that is strongly affiliated with another class.
  – can help increase encapsulation and reduce undesired name conflicts.
- an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure.

Insertion into Linked Lists

- Allocate new node
- Insert new element at the right place



Deletion is similar:

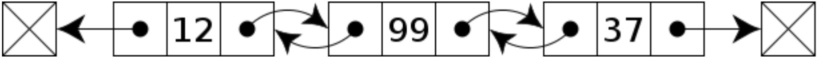


Special cases

- Inserting at the head
- Deleting at the head
- Inserting at the end
- Deleting at the end – There is no constant-time way to update the tail to point to the previous node

Doubly Linked Lists

- Nodes store: element, link to the previous node, link to the next node
- Special trailer and header nodes
- can be traversed forward and backward

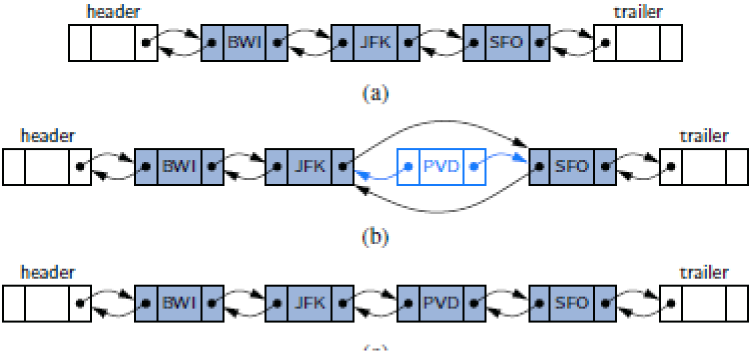


By Lasindi - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2245165

Doubly Linked Lists - Insertion

`addBetween(E e Node<E> predecessor, Node<E> succ`

(full implementation in Ch 3.4.1 of the text)



DLL - Insertion and Deletion

```
private void addBetween(E e, Node<E> predecessor,
        Node<E> successor) {
    Node<E> newest = new Node<>(e, predecessor, successor)
    predecessor.setNext(newest); /*link forward */
    successor.setPrev(newest); /* link back */
    size++;
}

private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> sucessor = node.getNext();
    predecessor.setNext(successor); /* link forward */
    successor.setPrev(predecessor); /* link back */
    size--;
    return node.getElement();
}
```

Doubly Linked Lists - Comments

- Insertion and Deletion of any given node takes O(1) time
- finding the insertion location or the node to delete may take longer!
- the special sentinel nodes are not required but make programming easier

The Stack Data Structure

- Useful in doing some computations, e.g., parenthesis matching, depth First Search in graphs
- Storing page-visited history in a Web browser
- Undo sequence in a text editor
- Used to implement recursion
- Used to classify languages (EECS 2001) - context-free languages

The Stack ADT

- stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations: push(e), pop()
- Other operations: object top(), integer size(), boolean isEmpty()

The Stack Interface

Different from the built-in Java class java.util.Stack

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty

```
public interface Stack<E> {
    int size();
    boolean isEmpty();
    E top();
    void push(E element);
    E pop();
}
```

Stack Operations

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

Implementing Stacks with Arrays

- We add elements from left to right
- A variable keeps track of the index of the top element

Limitations

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Implementing Stacks with Arrays - 2

```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        return null
    else
        t =t − 1
        return S[t + 1]

Algorithm push(o)
    if t = S.length − 1 then
        throw IllegalStateException
    else
        t = t + 1
        S[t] = o
```

Implementing Stacks with Arrays - Performance

Let n be the number of elements in the stack

- The space used is $\theta(n)$
- Each operation runs in time $O(1)$

Questions

Reversing Arrays: Write a program to reverse an array using a stack

Matching Parentheses: Write a program to math paraentheses in an expression using a stack

Stack class and Inheritance: Can you declare a stack of some class and insert instances of a subclass of it?

The Queue Data Structure

- Useful in doing some computations, e.g., breadth First Search in graphs, simulations
- Job scheduling
- Waiting lists, bureaucracy
- Multiprogramming

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations: enqueue(object), object dequeue()
- Other operations: object first(), integer size(), boolean isEmpty()
- dequeue or first on an empty queue returns null

The Queue ADT

Operation	*Output*	Q
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

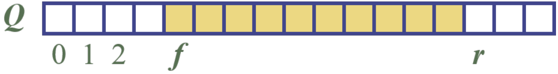
Implementing Queues with Arrays

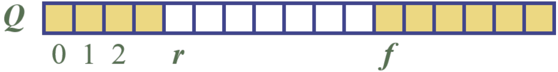
- Use an array of size N in a circular fashion
- Variables:

  f: index of the front element

  sz: number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \mod N$ is the first empty slot past the rear of the queue

Normal and Wrapped-around configurations:





Implementing Queues with Arrays - 2

```
Algorithm size()
    return sz

Algorithm isEmpty()
    return (sz == 0)

Algorithm enqueue(o)
    if size() = N − 1 then
        throw IllegalStateException
    else
        r = (f + sz) mod N
        Q[r] = o
        sz = sz + 1
```

Implementing Queues with Arrays - 3

```
Algorithm dequeue()
    if isEmpty() then
        return null /* Note: no exception */
    else
        o = Q[f]
        f = (f + 1) mod N
        sz = (sz - 1)
        return o
```

Implementing Stacks and Queues with Linked Lists

- Linked Lists are natural candidates for these uses
- Free from the size limitations of array based implementations
- Singly linked lists are sufficient for stacks – $\theta(1)$ push and pop operations
- Doubly linked lists are needed for efficient implementation of queues
- Enqueue: $\theta(1)$ on both, but dequeue: $\theta(1)$ on doubly linked lists, $\theta(k)$ on singly linked lists