CO322: DS & A
Linear Data Structures

Dhammika Elkaduwe

Department of Computer Engineering
Faculty of Engineering

University of Peradeniya

What is to come

- Look at some simple data structures
- Most of it you know and have used before :)
- Mostly a recap of what was discussed

Linked list

Limitation of arrays:

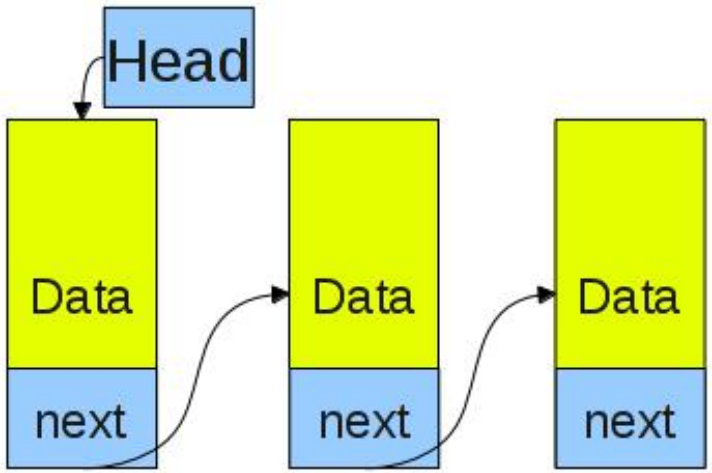
- Need to know the size at the start! (at least some good estimate)
- Can not go beyond that!

Advantages of arrays:

- Contiguous in memory (good cache locality if accessed sequentially)
- Very limited bookkeeping is required (only need to know the start of the array and its size)

Alternative $\Rightarrow$ Linked list Used when we do not know the size in advance and size varies a lot! (need to think before selecting)

Linked list ...



- Head points to the first element (ideal for sequential)
- Add/remove items as required
- Each element contains a pointer to the next
- No limit on how many elements we can have
- Additional storage for pointers

In your program you need to store, on an average N integers. In the worst case this can be 2N. Should you use an array or linked list? Explain. (assume 32bit addresses).

Linked list in Java

```
import java.util.*;

class Linked {
    public static void main(String [] args) {
LinkedList<String> list = new LinkedList<String>();

for(int i=0; i<10; i++)
    list.add(Integer.toString(i));

list.remove();

System.out.println("List content: "+ list);
    }
}
```

```
$ java Linked
List content: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Linked list in Java interface

from:
http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

- add(E e)
- add(int index, E e)
- addFirst(E e)
- contains(E e)
- get(int index)
- ...

Note: Somewhat advanced than the simple structure we discussed. Stemming from programmer needs! You might need different representations to support implementing the above interface (example: two pointers to find head and tail.)

Linked list variations

Some issues of *singly-linked list* and variations to resolve issues:

- Difficult to remove from the middle (you know the element, want ro remove)
  - Doubly-linked list: Each node has two pointers: next and previous.
- You need to worry about corner cases
  - Circular-linked list: Last node points to the first node
- Combination of both variations: Circular-doubly-linked list

Queue data structure

Basic idea:

- First-in First-out (FIFO) data structure
- Linked list with tail insert and head removal

  should we have a doubly-linked list for this?
  should we have a circular-linked list for this?
- Or can be implemented using an array

Queue operations:

- add(E e)
- remove(E e)
- peek()

Used when every you need FIFO processing.
Think of some Examples?

Example application: background code

```
class Student {
  public String eNumber, fieldOfChoice;
  public double gpa;
  Student(String eNumber, String fieldOfChoice, double
      gpa) {
    this.eNumber      = eNumber;
    this.fieldOfChoice = fieldOfChoice;
    this.gpa          = gpa;
  }
  public void display() {
    System.out.println(this.eNumber + ": GPA = " +
        this.gpa + " wants to do " + this.fieldOfChoice);
  }
  public boolean preferComp() {
    return new
        String("Computer").equals(this.fieldOfChoice);
  }
}
```

Example application: background code

```
import java.util.*;

class QueueEx {

  static String [] fields = {"Chemical", "Civil",
      "Computer",
    "Electrical", "Mechanical", "Production"};
  static int comp = 5;

  static String getFieldOfChoice() {
    return fields[(int)(Math.random() * 10 % 6)];
  }

 static double getGPA() {
   return Math.random() * 4 ;
 }
 /* more to come */
```

Example application: main

```
public static void main(String [] args) {
  Queue<Student> queue = new LinkedList<Student>();
  Student s;
  for(int i=0; i<400; i++) {
    s = new Student("E/11/0" + Integer.toString(i),
    getFieldOfChoice(),
    getGPA());
    queue.add(s);
  }
  while(!queue.isEmpty()) { /* note isEmpty */
    s=queue.remove();
    if((s.preferComp()) && (comp-- > 0)) {
      s.display();
    } /* while */ } /* main ends */
} /* end of class */
```

Broken code! We do not need FIFO; need priority queue

Priority Queues

Oder or removal depends on priority

- Internally the list is kept sorted. When inserting (or adding) a new element add to it's correct place. (recall insertion sort)(insert $O(N)$, removal $O(1)$)
- Else we can insert to the tail and locate the highest element when removing? (insert $O(1)$, removal $O(N)$)
- Which method is better? Why?

Priority Queues in Java

```
class StudentComp implements Comparator<Student> {
  public int compare(Student a, Student b) {
    return Double.compare(b.gpa, a.gpa); /* since we want
        descending */
  }
}
```

```
/* part of the main method. Whole main is in the next
    slide
*/
  PriorityQueue<Student> queue = new
    PriorityQueue<Student>(2, new StudentComp());
          ..........
```

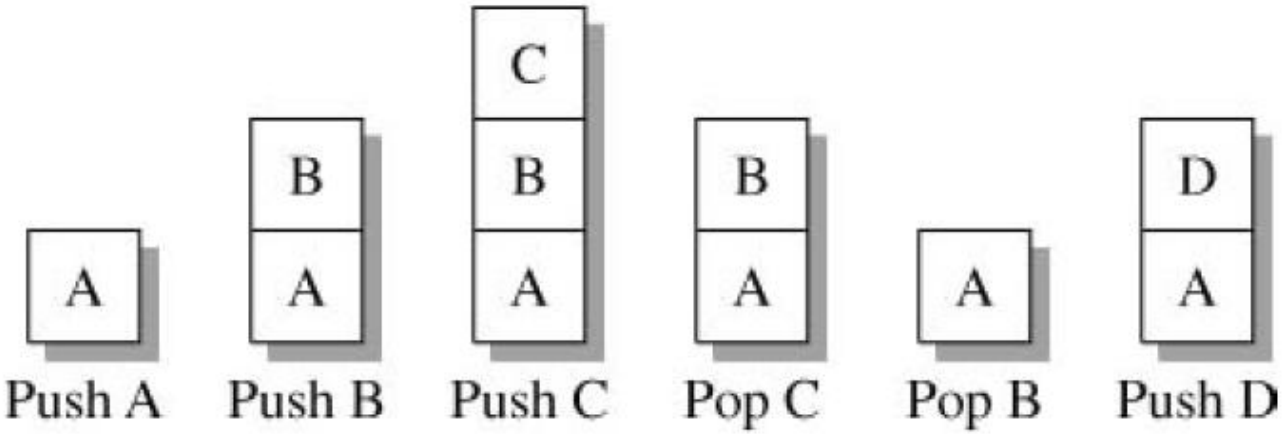
Priority Queues Example

```
public static void main(String [] args) {
  PriorityQueue<Student> queue = new
      PriorityQueue<Student>(2, new StudentComp());
  Student s;

  for(int i=0; i<400; i++) {
    s = new Student("E/11/0" + Integer.toString(i),
        getFieldOfChoice(), getGPA());
    queue.add(s);
  }

  while(!queue.isEmpty()) {
    s=queue.remove();
    if((s.preferComp()) && (comp-- > 0)) {
      s.display();
    }
  }
}/* main */
```

The stack data structure

Basic idea:

- *"list of items that are accessible from only one end"*
- Last-In First-Out (**LIFO**) data structure
- Two main operations: push and pop
- Other operations: isEmpty, peek, isFull ..
- Can be implemented using arrays or linked lists



Stack application

```
int factorial(int x) {
  if(x == 1) return 1;
  return x * factorial(x-1);
}
```

- A stack is used to postpone obligations, which must later be fulfilled in reverse order.
- Recall implementing *tower of Hanoi* using non-recursive implementation.

Stack: Java implementation

```
public interface Stack{
  void push(Object item);
  Object pop();
  boolean isEmpty();
  /* the interface can be implemented in many ways */
}
```

Stack: Implementation of interface

```
public class StackArray implements Stack {
  private Object [] array;
  private int size = 0,capacity = 5;

  public StackArray() {
    array = new Object [capacity];
  }

  public void push(Object item) {
    if(size < capacity) array[size++] = item;
    else throw new IllegalStateException("Full array");
  }
  public Object pop() {
    if(size != 0) return array[--size];
    return null;
  }
  public boolean isEmpty() { return (size == 0); }
}
```

Stack: Implementation of interface

```
public class StackList implements Stack {
  private class Node {
    public Object item;
    public Node next;
    public Node(Object item, Node next) { /* item to add
        list */
      this.item = item;
      this.next = next;
    }
  } /* Node class */
```

Stack: Implementation of interface cnt..

```
  private Node top; /* stack top */

  public StackList() { top = null; }
  public void push(Object item) {
    top = new Node(item, top);
  } /* end push */
  public Object pop() {
    if(!isEmpty()) {
      Object t = top.item;
      top = top.next;
      return t;
    }
    return null;
  } /* end pop */
  public boolean isEmpty() { return top == null; }
}
```

Use of a stack: Example 1: Converting to binary

Convert the first argument into binary:

```
class Binary {
  public static void main(String [] args) {
    int rem, a = Integer.parseInt(args[0]);
    Stack s = new StackList();
    while(a >= 1) {
      rem = a % 2;
      s.push(rem);
      a = a / 2;
    }
    while(!s.isEmpty())
    System.out.print(s.pop());
  }
}
```

Use of a stack: Example 2: Check brackets

Algorithm:

- Scan the string from left to right
- When you find an opening bracket push it
- When you find closing bracket pop and see if it matches
- At the end of the string, stack should be empty.

```
public static boolean checkBalance(String str) {
  Stack s = new StackList();
  char [] array = str.toCharArray();
  /* implement the rest */
```

Use of a stack: Example 2: Check brackets code

```
public static boolean checkBalance(String str) {
  Stack s = new StackList();
  char [] array = str.toCharArray();

  for(int i=0; i < array.length; i++) {
    if(array[i] == '[' || array[i] == '('
        s.push(array[i]);


      if(array[i] == ']') if(s.pop() != '[') return false;
      if(array[i] == ')') if(s.pop() != '(') return false;
    } /* for( */

    return s.isEmpty();
  }
```

What is the run-time complexity of this code? (N char and M bracket types)

Different types of expressions

Expression type	Example
Postfix	a b +
Infix	a + b
Prefix	+ a b

- In a postfix evaluation format for an arithmetic expression, an operator comes after its operands.
- In a infix evaluation format for an arithmetic expression, an operator comes between its operands.
- In a prefix evaluation format for an arithmetic expression, an operator comes before its operands.

Infix to postfix

Algorithm:

- Read the expression from left to right
- If you find a operator push to stack until the operand has less precedence than the one on top of the stack.
- When the operator has less precedence pop till stack is empty or operator has higher precedence
- Operands are put as is.
- If the stack is not empty, pop them.

Expression	Output	Stack
A + B * C - D		
A + B * C - D	A	
A + B * C - D	A B	+
A + B * C - D	A B C	+ *
A + B * C - D	A B C * +	-
A + B * C - D	A B C * + D -	

Evaluation of a postfix expression

Algorithm:

- Read the expression from left to right
- push the operands to stack
- when you find an operator, pop two operands from the stack, do the operating and push the result.
- At the end of the expression, stack should have only one value which is the evaluation of the expression.

Original expression: A + B * C - D

Expression	Stack	Current Operation
A B C * + D -		
A B C * + D -	A B C	
A B C * + D -	A	B * C
A B C * + D -	A (B * C)	
A B C * + D -	A + (B * C)	
A B C * + D -	A + (B * C) D	
A B C * + D -		A + (B * C) - D

Exercises

1. Implement a Java function *int evaluate(String postfixExpression)* which will evaluate the given postfix expression
2. Implement a Java function *String evaluate(String postfixExpression)* which will convert the infix expression into postfix.