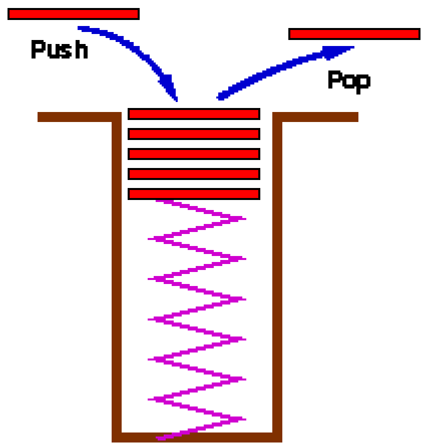
2. Stacks, Queues and Hashing

2.1. Stacks

- A stack is a data structure in which all the access is restricted to the most recently inserted items.
- If we remove this item, then we can access the next-to-last item inserted, and etc.
- A stack is also a handy aid for algorithms applied to certain complex data structures.

Stack Model contd…

- Input to a stack is by push
- Access is by top
- Deletion is by pop



Example

- Stack contains 3, 4
- Push item 9
  - Now the stack contains 3,4,9
- If we pop now, we get 9
- If we pop again, we get 4

Stack Properties

- The last item added to the stack is placed on the top and is easily accessible.
- Thus the stack is appropriate if we expect to access on the top item, all other items are inaccessible.

2.1.3. Important stack applications

- Compiler Design
- Mathematical Expression Evaluation
- Balanced Spell Checker
- Simple Calculator

The stack operations

- ***clear()*** – Clear the stack
- ***isEmpty()*** – Check to see if the stack is empty
- ***push(el)*** – Put the element *el* on top of the stack.
- ***pop()*** – Take the topmost element from stack.
- ***topEl()*** – Return the topmost element in the stack without removing it.

2.1.1. Implementation Of Stacks

- There are two basic ways to arrange for constant time operations.
  - The first is to store the items contiguously in an array.
  - And the second is to store items non-contiguously in a linked list

Array Implementation

- To push an item into an empty stack, we insert it at array location 0. (since all java arrays start at 0)
- To push the next item into the location 0 over the location to make room for new item.

Array Implementation contd…

- This is easily done by defining an auxiliary integer variable known as stack pointer, which stores the array index currently being used as the top of the stack.
- A stack can be implemented with an array and an integer.
- The integer *TOS* (top of stack) provides the array index of the top element of the stack, when *TOS* is –1 , the stack is empty.

Stacks specifies two data fields such as

- The array (which is expanded as needed stores the items in the stack)

- TopOfStack (*TOS*) (gives the index of the current top of the stack, if stack is empty, this index is –1.)

Algorithms For Pushing & Popping

PUSH

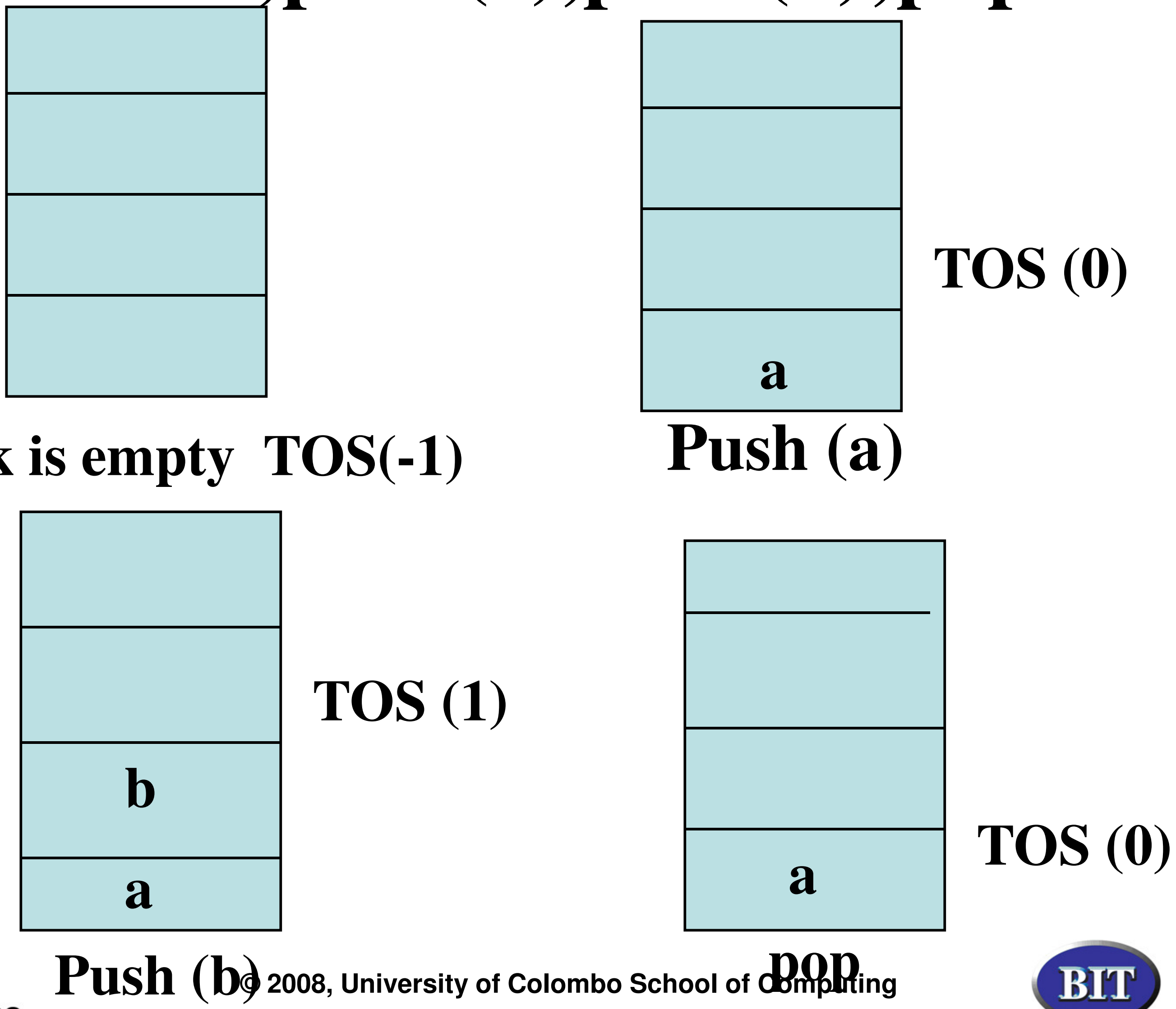
- If stack is not full then
- Add 1 to the stack pointer.
- Store item at stack pointer location.

Algorithms For Pushing & Popping contd…

POP

- If stack is not empty then
- Read item at stack pointer location.
- Subtract 1 from the stack pointer.

How to stack routines work:empty stack;push(a),push(b);pop



Stack is empty TOS(-1)

Push (a)

Push (b)

pop

Java implementation

- Zero parameter constructor for array based stack

```
public stackar( )
/* construct the stack
{
 thearray= new object[default-capacity];
Tos = -1;
}
```

Isempty : returns true if stack is empty, false otherwise

- **Isempty( )**

```
public Boolean Isempty( )
{
return tos= = -1
}
```

Isfull( ) : returns true if stack is full , false otherwise

- Isfull( )

```
public Boolean isfull( )
{
return tos= = stacksize-1; (default capacity)
}
```

push method for array based stack

- Insert a new item into the stack

```
public void push (object x)
{
If isfull( )
throw new stackexception (“ stack is full”)
theArray[++tos]=x;
}
```

Pop method for array based stack

- Remove the most recently inserted item from the stack
- Exception underflow if the stack is empty

```
public void pop( ) throws underflow
{
If (isempty(  ))
throw new underflow (“stackpop”);
tos - - ;
}
```

Top method for array based stack

- Return the most recently inserted item from the stack
- Exception underflow if the stack is empty

```
public object top( ) throws underflow
{
if (isEmpty( ))
throw new underflow( “stacktop”)
return theArray[tos];
}
```

Top and pop method for array based stack

- Return and remove the most recently inserted item from the stack
- Exception underflow if the stack is empty

```
public object topandpop( ) throws underflow
{
if isEmpty( ) )
throw new underflow (“stack topandpop”);
return theArray[tos - - ];
}
```

Java Code for a Stack

```
import java.io.*; // for I/O

class StackX
{
private int maxSize; // size of stack array
private double[ ] stackArray;
private int top; // top of stack
//--------------------------------------------------------------

public StackX(int s) // constructor
{
maxSize = s; // set array size
stackArray = new double[maxSize]; // create array
top = -1; // no items yet
}
//--------------------------------------------------------------
```

Java Code for a Stack

```
public void push(double j) // put item on top of stack {
stackArray[++top] = j; // increment top, insert item
}
//-----------------------------------------------------------

public double pop() // take item from top of stack {
return stackArray[top--]; // access item, decrement top
}
//-----------------------------------------------------------

public double peek() // peek at top of stack {
return stackArray[top];
}
//-----------------------------------------------------------
-
public boolean isEmpty() // true if stack is empty
{
return (top == -1);
}
```

Java Code for a Stack

```
//--------------------------------------------------------------
-
public boolean isFull() // true if stack is full
{
return (top == maxSize-1);
}
//--------------------------------------------------------------
-
} // end class StackX
```

Java Code for a Stack

```
class StackApp {
public static void main(String[] args)
{
StackX theStack = new StackX(10); // make new stack
theStack.push(20); // push items onto stack
theStack.push(40);
theStack.push(60);
theStack.push(80);
while( !theStack.isEmpty() ) // until it's empty,
{ // delete item from
stack double value = theStack.pop();
System.out.print(value); // display it
System.out.print(" ");
} // end while
System.out.println("");
} // end main()
} // end class StackApp
```

Java Code for a Stack

- The main() method in the StackApp class creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it's empty.
- Here's the output:

```
80 60 40 20
```

StackX Class Methods

- The constructor creates a new stack of a size specified in its argument.
- The fields of the stack comprise a variable to hold its maximum size (the size of the array), the array itself, and a variable top, which stores the index of the item on the top of the stack.

  (Note that we need to specify a stack size only because the stack is implemented using an array. If it had been implemented using a linked list, for example, the size specification would be unnecessary.)

StackX Class Methods

- The **push()** method increments top so it points to the space just above the previous top, and stores a data item there. Notice that **top** is incremented before the item is inserted.

- The **pop()** method returns the value at top and then decrements **top**. This effectively removes the item from the stack; it's inaccessible, although the value remains in the array (until another item is pushed into the cell).

- The **peek()** method simply returns the value at top, without changing the stack.

- The **isEmpty()** and isFull() methods return true if the stack is empty or full, respectively. The top variable is at –1 if the stack is empty and **maxSize-1** if the stack is full.

Error Handling

- There are different philosophies about how to handle stack errors. What happens if you try to push an item onto a stack that's already full, or pop an item from a stack that's empty?
- We've left the responsibility for handling such errors up to the class user. The user should always check to be sure the stack is not full before inserting an item:

Error Handling

```
if( !theStack.isFull() )
    insert(item);
else
    System.out.print("Can't insert, stack is full");
```

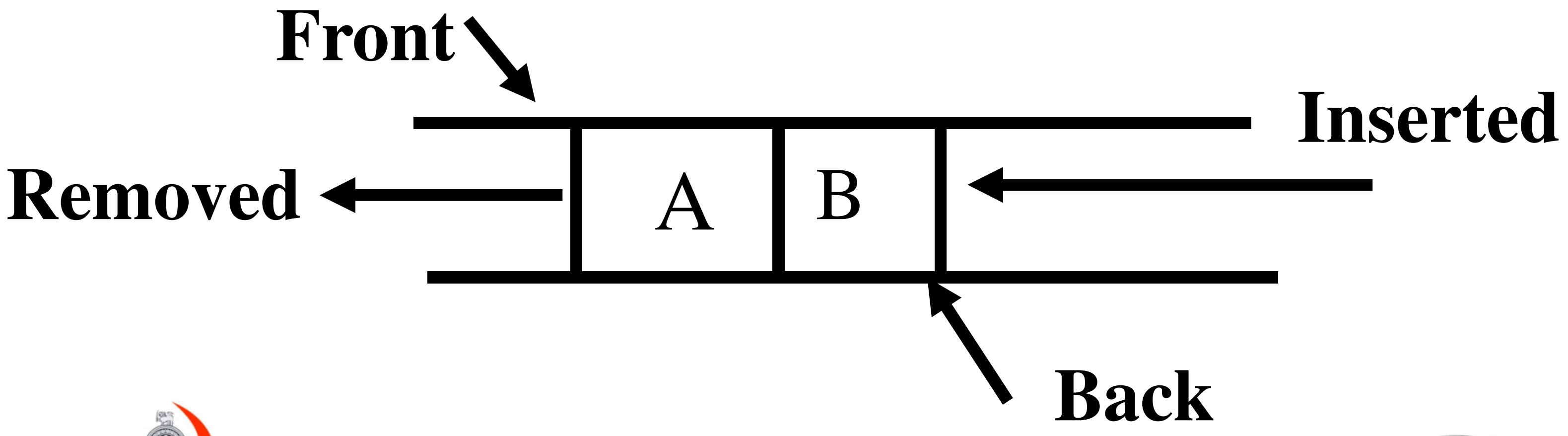
• In the interest of simplicity, we've left this code out of the main() routine (and anyway, in this simple program, we know the stack isn't full because it has just been initialized).
• We do include the check for an empty stack when main() calls pop().

2.1.2. Efficiency of Stacks

- Items can be both pushed and popped from the stack implemented in the StackX class in constant O(1) time.
- That is, the time is not dependent on how many items are in the stack, and is therefore very quick.
- No comparisons or moves are necessary.

2.2. Queues

- A *queue* is a special kind of list
- Items are inserted at one end (the *rear*) and (enqueue)
- deleted at the other end (the *front*). (dequeue)
- Queues are also known as 'FIFO lists'



2.2. Queues

- There are various queues quietly doing their job in our computer's (or the network's) operating system.
- There's a printer queue where print jobs wait for the printer to be available.
- A queue also stores keystroke data as we type at the keyboard.
- This way, if we are using a word processor but the computer is briefly doing something else when we hit a key, the keystroke won't be lost; it waits in the queue until the word processor has time to read it.
- Using a queue guarantees the keystrokes stay in order until they can be processed.

2.2.1. The Queue operations

- ***clear()*** – Clear the queue
- ***isEmpty()*** – Check to see if the queue is empty
- ***enqueue(el)*** – Put the element *el* on top of the queue.
- ***dequeue()*** – Take the first element from queue.
- ***firstEl()*** – Return the first element in the queue without removing it.

2.2.2. A Circular Queue

- When we insert a new item in the queue in the Workshop applet, the Front arrow moves upward, toward higher numbers in the array.
- When we remove an item, Rear also moves upward.
- we may find the arrangement counter-intuitive, because the people in a line at the movies all move forward, toward the front, when a person leaves the line.
- We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient.
- Instead we keep all the items in the same place and move the front and rear of the queue.

2.2.2. A Circular Queue contd…

- To avoid the problem of not being able to insert more items into the queue even when it's not full, the Front and Rear arrows *wrap around* to the beginning of the array. The result is a *circular queue*

2.2.3. Java Code for a Queue

- The queue.java program features a Queue class with insert(), remove(), peek(), isFull(), isEmpty(), and size() methods.
- The main() program creates a queue of five cells, inserts four items, removes three items, and inserts four more. The sixth insertion invokes the wraparound feature. All the items are then removed and displayed.
- The output looks like this:

```
40 50 60 70 80
```

The Queue.java Program

```
import java.io.*; // for I/O
class Queue {
private int maxSize;
private int[] queArray;
private int front;
private int rear;
private int nItems;
//-------------------------------------------------------------

public Queue(int s) // constructor
{
maxSize = s;
queArray = new int[maxSize];
front = 0;
rear = -1;
nItems = 0;
}
//-------------------------------------------------------------
```

The Queue.java Program

```
public void insert(int j) // put item at rear of queue
{
if(rear == maxSize-1) // deal with wraparound
rear = -1;
queArray[++rear] = j; // increment rear and
insert
nItems++; // one more item
}
//--------------------------------------------------------------
-
public int remove() // take item from front of queue
{
int temp = queArray[front++]; // get value and incr front
if(front == maxSize) // deal with wraparound
front = 0;
nItems--; // one less item
return temp;
}
//--------------------------------------------------------------
```

The Queue.java Program

```
public int peekFront() { // peek at front of queue
return queArray[front];
}
//--------------------------------------------------------------
public boolean isEmpty() { // true if queue is empty
return (nItems==0);
}
//--------------------------------------------------------------
public boolean isFull() { // true if queue is full
return (nItems==maxSize);
}
//--------------------------------------------------------------
public int size() { // number of items in queue
return nItems;
}
//--------------------------------------------------------------
} // end class Queue
```

The Queue.java Program

```
class QueueApp {
public static void main(String[] args) {
Queue theQueue = new Queue(5); // queue holds 5 items
theQueue.insert(10); // insert 4 items
theQueue.insert(20);
theQueue.insert(30);
theQueue.insert(40);
theQueue.remove(); // remove 3 items
theQueue.remove(); // (10, 20, 30)
theQueue.remove();
theQueue.insert(50); // insert 4 more items
theQueue.insert(60); // (wraps around)
theQueue.insert(70);
theQueue.insert(80);
while( !theQueue.isEmpty() ) { // remove and display all items
int n = theQueue.remove(); // (40, 50, 60, 70, 80)
System.out.print(n);
```

The Queue.java Program

```
System.out.print(" ");
}
System.out.println("");
} // end main()
} // end class QueueApp
```

2.2.3.1. Efficiency of Queues

- As with a stack, items can be inserted and removed from a queue in O(1) time.

2.2.3.2. Deques

- A deque is a double-ended queue.
- We can insert items at either end and delete them from either end.
- The methods might be called insertLeft() and insertRight(), and removeLeft() and removeRight().
- If we restrict ourself to insertLeft() and removeLeft() (or their equivalents on the right), then the deque acts like a stack.
- If we restrict ourself to insertLeft() and removeRight() (or the opposite pair), then it acts like a queue.
- A deque provides a more versatile data structure than either a stack or a queue, and is sometimes used in container class libraries to serve both purposes.
- However, it's not used as often as stacks and queues, so we won't explore it further here.

2.2.4. Priority Queues

- A priority queue is a more specialized data structure than a stack or a queue.
- However, it's a useful tool in a surprising number of situations.
- Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front.
- However, in a priority queue, items are ordered by key value, so that the item with the lowest key (or in some implementations the highest key) is always at the front.
- Items are inserted in the proper position to maintain the order.

2.2.4.1. Efficiency of Priority Queues

- In the priority-queue implementation we show here, insertion runs in O(N) time, while deletion takes O(1) time.

2.3. Hash Functions

- Choosing a good hashing function, **h(k)**, is essential for hash-table based searching.
- The key criterion is that there should be a minimum number of collisions.
- *Sophisticated hash functions* – may avoid collisions,computational cost in determining h(k) can be prohibitive.
- Less sophisticated methods may be faster.

2.3. Hash Functions contd…

- The number of hash functions that can be used to assign positions to n items in a table of m positions (for n <= m) is equal to m^n.

- Some specific types of hash functions are:
  - **Division**

    h(k) = K mod TSize *Tsize = sizeof(table)*

    Usually the preferred choice for the hash function if very little is known about the keys.

2.3. Hash Functions contd…

– **Mid-Square Function**

The key is squared and the middle or mid part of the result is used as the address. If the key is a string, it has to be pre-processed to produce a number.

e.g. if the key is 3121 then $3121^2 = 9740641$ and h(3121) = 406.

2.3. Hash Functions contd…

– **Extraction**

Only a part of the key is used to compute the address. For the key 123-45-6789 this method might use the

first four digits 1234

last four digits 6789

the first two combined with last two 1289 or

some other combination

2.3. Hash Functions contd…

- Radix Transformation

The key is transformed into another number base.

e.g. If K is the decimal number 345, then its value in base 9 is 423. This value is then divided modulo TSize, and the resulting number is used as the address of the location to which K should be hashed.