Chapter 3- Stacks and Queues

3.1 Definition of Stack Data Structure and its operations

3.2 Application of Stacks

3.3 Array Implementation of Stack

3.4 Linked List Implementation of Stack

3.5 Queues and Operations on Queues

3.6 Array and Linked List implementation of Queues

3.7 Stacks and Queues in JDK

3.8 Exercises

3.1- Definition of Stack Data Structure and its operations

3.1 Stacks

- A stack is an arrangement of objects such that last that gets added comes off first.
- Stack has a LIFO (last in first out) philosophy
- Consider a stack of books for example, when we add a book, we add to the top of stack, and when we remove a book, it is easy to remove the one we just added to the top.
- Stacks can be used to reverse a set of items.

3.1 Operations on Stacks

The operations that are required to be implemented are:

public Stack() // initializes the stack
pubic boolean empty() // returns true if the stack is full, false otherwise
public void push(T item) // adds *item* to the top of stack.
public T pop() // returns top most item and removes it
public T peek() //returns the top most item but does not remove it.

3.1 Operations on Stacks

Note that a stack can never get full. Therefore we need to extend capacity of the stack just like we did in the ArrayADT class.

Also note that there is no *iterator()* for the StackADT. This is the norm since Stack is not a list ADT

Some implementations provide *size(), get(i)* and *toString()* functions for the Stack. But we will provide only O(1) operations for the stack.

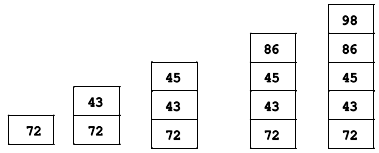
3.2 Application of Stacks

3.2 Reverse using Stacks

Let us say we want to reverse the following numbers using stacks.

2 43 45 86 98

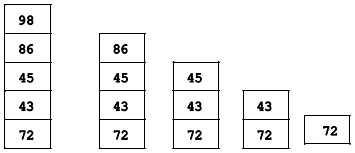
We add one by one to the stack, by pushing each item onto the stack



push(72) push(43) push(45)
push(86) push(98).

3.2 Reverse Using Stacks

Now we pop off elements one by one to get it in the reverse order.



pop() pop() pop() pop()

98 86 45

3.2 Reverse with Stacks

Using stacks, one can write a program to reverse a string ending with a '.'

```
public void reverse(String s){
 Stack<Character> st = new Stack();
       int i = 1;
      Character letter = new Character(s.charAt(0);
       while (letter.equals('.')){
           st.push(letter);
           letter = new Character(s.charAt(i++);
       }
        while (!st.empty()){
              letter = pop(S);
              System.out.print(letter);
        }
}
```

3.2 Delimiter Matching using Stacks

In an expression like {3+4[5+(6*(9-4))] + (12 +x*[5 4])}*{4+5},

Suppose we want to make sure that the expressions has matched delimiters. We can use stacks data structure for this purpose. The algorithm is simple:

- Process the input from left to right
- Whenever a delimiter is seen, if it is a opening delimiter like "(" or "[" or "{" then push it on the stack
- Whenever a delimiter is seen, if it is a closing delimiter like ")" or "}" or "]" then pop the corresponding opening delimiter from stack. If there is no corresponding opening delimiter raise an error condition.
- When the input is done, the stack should be empty.

3.2 In Class Activity

Evaluate the following expressions for matching delimiters

{3+4[5+(6*(9-4))] + (12 +x*[5-4])}*{4+5},

3 + {4 + 5(8+5) (v+x})

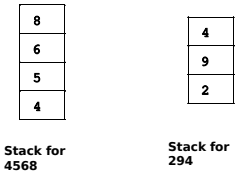
3.2 In Class Activity

Write a method for delimiter matching in an expression using stack operations

3.2 Large Integer Addition using Stacks

Really large integers can be added using stack data structure. Here is the process.

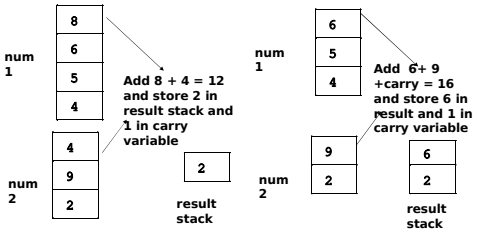
Suppose the integers are 4568 and 294, then we first make two stacks each containing the digits in the numbers and accumulate the result 4862



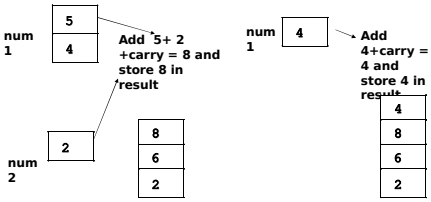
Stack for 4568

Stack for 294

3.2 Large Integer Addition Using Stacks



3.2 Large Integer Addition using Stacks



Now we pop the result stack to get the sum 4862.

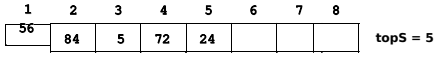
3.2 In Class Activity

Write a method for large integer addition using stacks.

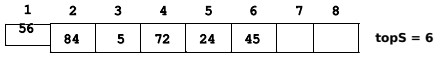
3.3 Array Implementation of Stacks

3.3 Array Implementation of Stacks

Assume stack will be an array. Assume *topS* will be the position of the top element.

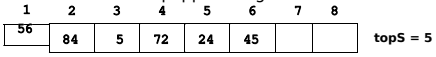


When an element is *pushed,* we put it in the next position available and increase *topS* by 1. For example, when 45 is *pushed*, we get



3.3 Array Implementation of Stacks

When a *pop* is requested, we remove the element at
topS and return it. We decrease *topS* by 1.
When the stack is popped we get 45.



full() is implemented by checking if topS = Size of array. and *empty()* is implemented by checking if topS = ?

peek() is implemented by simply returning the element at topS but topS or the array is unchanged.

3.3 Array Implementation of Stacks

```
public class Stack{
   int capacity = 100;
   T[] elems;
   int topS;

   pubic Stack(){
     elems = new T[capacity];
     topS = -1;
  }
```

3.3 Array Implementation of Stacks

```
public void push(T item){
         if (topS == capacity -1)
             extendCapacity(item);
         elems[topS] = item;
         tops++;
}

public T pop() throws EmptyStackException{
        if (empty())
         throw (new EmptyStackException());
        else{
         tops--;
         return(elems[tops-1]);
        }
}
```

3.3 Array Implementation of Stacks

```
  public boolean empty(){
    return(topS == -1);
  }

  public boolean full(){

  }
  public boolean empty(){
    return(topS == -1);
  }

  private void extendCapacity(T item){
   // write this on your own
  }
```

3.4 Linked List Implementation of Stacks

3.4 Linked List Implementation of Stacks

Write a linked list implementation of Stacks on your own. Use the methods from LinearNode class we wrote earlier. Do not use the methods in the LinkedList class (why?)

3.5 Queues and Operations on Queues

3.5 Queues

- A queue is an arrangement of objects such that first item that gets added comes off first.
- Queue has a FIFO (first in first out) philosophy

Queues have natural application whenever we want to process data in a first come first served basis.

-- queue of programs to be run by a operating system
-- queue of jobs to be printed from printer.
-- queue of people to be server at a bank teller etc.

3.5 Operations on Queues

Following operations are provided for QueueADT.

public Queue() **//initializes the queue**

public boolean empty() **// returns true if the queue is empty, false otherwise**

public boolean deque() **//returns and removes the item at the head of the queue**

public boolean enque(T item) **//adds item to that tail of the queue.**

3.5 Operations on Queues

Note that a Queue can never get full.

Therefore we need to extend capacity of the stack just like we did in the ArrayADT class.

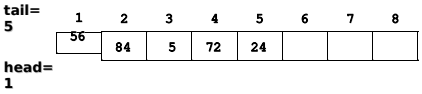
Also note that there is no *iterator()* for the Queue ADT. This is the norm since Stack is not a list ADT

Some implementations provide *size(), get(i)* and *toString()* functions for the Queue. But we will provide only O(1) operations for the stack.

3.6 Array and Circular Array Implementation of Queues

3.6 Implementation of Queues using arrays

We need to keep track of both *head* and the *tail* of queue. New elements will be *enqued* at *tail* and elements will be *dequed* from *head*.



3.6 Implementation of Queues using Arrays

When we issue a *deque*, the number *56* will be returned and the array will be

tail= 5

1	2	3	4	5	6	7	8
	84	5	72	24			

head= 2

When we issue a *enque(67)* the number *67* will be added at the tail of the queue

tail= 6

1	2	3	4	5	6	7	8
	84	5	72	24	67		

head= 2

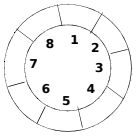
3.6 Implementation of Queues Using Arrays

What is the maximum number of elements that can be added to the queue, and is there an issue with this implementation?

3.6 Problem with the implementation

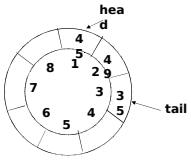
Use of linear array causes the problem, since the initial part of the array cannot be reused.

Use circular array. When the last position is used, use the first position again if it is empty



3.6 Circular Array Implementation of Queues

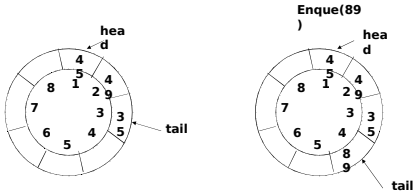
When an item is queued it is added at the tail, and when it is removed, it is removed from head.



When head(tail) has to move from 8 to 1, we use (head +1) mod 8.

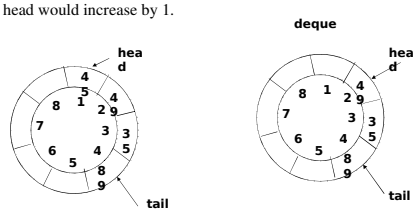
3.6 Circular Array Implementation of Queues

Enqueing 89 to the queue would result in *tail* being increased by 1 and the item put at the new *tail* index



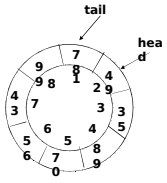
3.6 Circular Array Implementation of Queues

Dequeing the queue would result in 45 being returned and head would increase by 1.



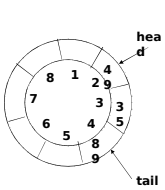
3.6 Circular Array Implementation of Queues

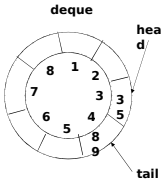
When is the queue full? When head = (tail +1) MOD 8



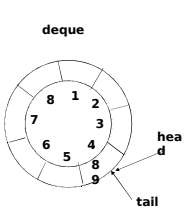
3.6 Circular Array Implementation of Queues

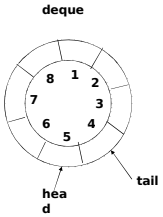
When is the queue empty? Lets remove the three elements.





3.6 Circular Array Implementation of Queues





When is the queue empty, head = (tail +1) MOD 8 !!!

3.6 Circular Array Implementation Of Queues

```
public class Queue{
     int capacity = 100;
     T[] elems;
     int head;
     int tail;
     int size;

   public Queue(){
        head = 0;
        tail = -1;
        elems = new T[capacity];
        size = 0;
   }
```

3.6 Circular Array Implementation Of Queues

```
public enque(T item){
      if (size == capacity)
          extendCapacity(item);
      else{
         elems[(tail+1) % capacity] = item;
         tail = (tail +1) %capacity;
         size++;
      }
}
```

3.6 Circular Array Implementation Of Queues

```
public deque()throws NoSuchElementException{
      if (empty())
         throw(new NoSuchElementException());
      else{
           tail = (tail -1) % capacity;
           size--;
           return(elems[(tail+1)%capacity];
      }
}
```

3.6 Circular Array Implementation Of Queues

```
public  boolean empty(){

      return(size ==0);
}

private void extendCapacity(T item){
     //write on your own
}
```

3.6 In Class Activity

Write a linked list implementation of Queues on your own. Use the methods from LinearNode class we wrote earlier. Do not use the methods in the LinkedList class.

3.6 Variations of Queues

Priority Queues: Queues in which the deque operation uses both priority of the item and the position of the item. Data can stored in decreasing or increasing priority.

How can priority queues be implemented?

These can be implemented with linked list but insert will take linear time.

3.7 Stacks and Queues in JDK

Stacks in java.util

Constructor List

public Stack()

Method List

public boolean empty()

Tests if this stack is empty.

public synchronized int search(Object o)

Returns where an object is on this stack.

Stacks and Queues in java.util

public synchronized Object peek()

Looks at the object at the top of this stack without removing it from the stack.

public synchronized Object pop()

Removes the object at the top of this stack and returns that object as the value of this function.

public Object push(Object item)

Pushes an item onto the top of this stack.

3.8 Exercises

3.8 Exercises

1. Explain the following terms in your own words:
   a. stack
   b. FIFO queue
   c. Priority queue
2. Give two everyday examples of each of the following:
   a. A stack
   b. A FIFO queue
   c.A priority queue
3. Write an operation called get (int i), which returns the value of the element that is the ith most recently arrived element. (The top element is at depth 1.) The stack remains unchanged.
4. Write a operation depth() which returns the current number of elements in the stack.
5. Queues are often specified with another operation – *cancel*. The idea is to be able to delete a given element from the queue. The element is identified by its unique key value:
   public void cancel (T item)
   Add the operation cancel for circular array implementation of queues and linked list implementation of queues.

3.8 Exercises

6. Implement Stack operations using arrays with first element of the array as topS.

7. Discuss implementation of queues using doubly linked lists.

8. Write an application to merge two stacks that have elements in sorted ascending order into a queue containing elements in sorted ascending order. What is the time complexity of your algorithm?

9. Write an application using stack ADT to reverse a given linked list of elements.

3.8 Exercises

10. Write an application to reverse the order of elements in a Stack
    a) Using two additional Stacks
    b) Use one additional Queue.
    c) Using one additional Stack and some variables.

11. Write an application to put the elements of stack in ascending order by using one additional stack and some additional variables.

12. Write an application using only additional variables, to put all the elements of a queue in ascending order,
    a) Using two additional queues
    b) Using one additional queue.