CS1083 Week 11: Stacks, Queues.

David Bremner

2018-03-21

Outline

Stacks

Queues

Stacks

Last In First Out list

Operations

required:	Push(object)
	Pop()
supporting:	Size()
	IsEmpty()
	Top()

Using a stack

LinkedStack

```
public static void main(String[] args){
    String sentence = "Shooby doo wop she bop" ;
    Scanner words =new Scanner(sentence);









}
```

Using a stack

LinkedStack

```
public static void main(String[] args){
    String sentence = "Shooby doo wop she bop" ;
    Scanner words =new Scanner(sentence);
    LinkedStack<String> stack=
        new LinkedStack<String>();









}
```

Using a stack

LinkedStack

```
public static void main(String[] args){
    String sentence = "Shooby doo wop she bop" ;
    Scanner words =new Scanner(sentence);
    LinkedStack<String> stack=
        new LinkedStack<String>();
    while(words.hasNext()){
        stack.push(words.next());
    }



}
```

Using a stack

LinkedStack

```
public static void main(String[] args){
    String sentence = "Shooby doo wop she bop" ;
    Scanner words =new Scanner(sentence);
    LinkedStack<String> stack=
        new LinkedStack<String>();
    while(words.hasNext()){
        stack.push(words.next());
    }
    while (!stack.isEmpty()){
        System.out.print(stack.pop()+" ");
    }
}
```

Adding to a stack

LinkedStack

```
public class LinkedStack<T> {

private Node<T> topNode=null;
```

```
public void push(T obj){
    Node<T> newNode=new Node<T>(obj);
    newNode.setNext(topNode);
    topNode=newNode;
}
```

Removing from a stack

LinkedStack

```
public T pop(){
    if (topNode==null)
        throw new EmptyStackException();

    T rval=topNode.getData();
    topNode=topNode.getNext();

    return rval;
}
```

Stacks to remember what to do

LinkedStack

```
class Coord {
public int r,c;
public Coord(int _r, int _c) {r=_r; c=_c;}
}
    ⋮
private LinkedStack<Coord> todo;
```

```
public boolean path(){
    todo= new LinkedStack<Coord>();
    todo.push(new Coord(startr,startc));









}
```

```
public boolean path(){
    todo= new LinkedStack<Coord>();
    todo.push(new Coord(startr,startc));
    while (!todo.isEmpty()){









    }
    return false;
}
```

```
public boolean path(){
    todo= new LinkedStack<Coord>();
    todo.push(new Coord(startr,startc));
    while (!todo.isEmpty()){
        Coord here=todo.pop();
        int r=here.r;
        int c=here.c;



    }
    return false;
}
```

```
public boolean path(){
    todo= new LinkedStack<Coord>();
    todo.push(new Coord(startr,startc));
    while (!todo.isEmpty()){
        Coord here=todo.pop();
        int r=here.r;
        int c=here.c;
        if (visit(r-1,c) || visit(r,c-1) ||
            visit(r+1,c) || visit(r,c+1))
            return true;
    }
    return false;
}
```

```
private boolean visit(int i, int j){
    if (i<0 || i>n || j<0 || j>m)
        return false;









    return false;
}
```

```
private boolean visit(int i, int j){
    if (i<0 || i>n || j<0 || j>m)
        return false;

    if(map[i][j] == Cell.FINISH)
        return true;




    return false;
}
```

```
private boolean visit(int i, int j){
    if (i<0 || i>n || j<0 || j>m)
        return false;

    if(map[i][j] == Cell.FINISH)
        return true;
    if(map[i][j] == Cell.PATH){



    }
    return false;
}
```

```
private boolean visit(int i, int j){
    if (i<0 || i>n || j<0 || j>m)
        return false;

    if(map[i][j] == Cell.FINISH)
        return true;
    if(map[i][j] == Cell.PATH){
        map[i][j]=Cell.EXPLORED;
        todo.push(new Coord(i,j));
    }
    return false;
}
```

Stacks

Queues

Queue

First In First Out list

Operations

required: Enqueue(object)
Dequeue ()

supporting: Size()
IsEmpty()
Front()

Queue

First In First Out list

Operations

required: Enqueue(object)
Dequeue ()

supporting: Size()
IsEmpty()
Front()

Queue

First In First Out list

Operations

required: Enqueue(object)
Dequeue ()

supporting: Size()
IsEmpty()
Front()

Queue

First In First Out list

Operations

required:
- Enqueue(object)
- Dequeue ()

supporting:
- Size()
- IsEmpty()
- Front()

Queue

First In First Out list

Operations

required:	Enqueue(object)
	Dequeue ()
supporting:	Size()
	IsEmpty()
	Front()

Queue Interface

```
public interface Queue<T>{
    // accessor methods




    // update methods




}
```

Queue Interface

```
public interface Queue<T>{
    // accessor methods
    public int size();
    public boolean isEmpty();


    // update methods




}
```

Queue Interface

```
public interface Queue<T>{
    // accessor methods
    public int size();
    public boolean isEmpty();
    public T front()
        throws QueueEmptyException;
    // update methods




}
```

Queue Interface

```
public interface Queue<T>{
    // accessor methods
    public int size();
    public boolean isEmpty();
    public T front()
        throws QueueEmptyException;
    // update methods
    public void enqueue (T element)
        throws QueueFullException;


}
```

Queue Interface

```
public interface Queue<T>{
    // accessor methods
    public int size();
    public boolean isEmpty();
    public T front()
        throws QueueEmptyException;
    // update methods
    public void enqueue (T element)
        throws QueueFullException;
    public T dequeue()
        throws QueueEmptyException;
}
```

Using a Queue

LinkedQueue

```
LinkedStack<String> stack=
    new LinkedStack<String>();









while (!stack.isEmpty()){



}
```

Using a Queue

LinkedQueue

```
LinkedStack<String> stack=
    new LinkedStack<String>();
Queue<String> queue=
    new LinkedQueue<String>();




while (!stack.isEmpty()){


}
```

Using a Queue

LinkedQueue

```
LinkedStack<String> stack=
    new LinkedStack<String>();
Queue<String> queue=
    new LinkedQueue<String>();
while(words.hasNext()){
    String word=words.next();

}
while (!stack.isEmpty()){


}
```

Using a Queue

LinkedQueue

```
LinkedStack<String> stack=
    new LinkedStack<String>();
Queue<String> queue=
    new LinkedQueue<String>();
while(words.hasNext()){
    String word=words.next();
    stack.push(word); queue.enqueue(word);
}
while (!stack.isEmpty()){



}
```

Using a Queue

LinkedQueue

```
LinkedStack<String> stack=
    new LinkedStack<String>();
Queue<String> queue=
    new LinkedQueue<String>();
while(words.hasNext()){
    String word=words.next();
    stack.push(word); queue.enqueue(word);
}
while (!stack.isEmpty()){
    System.out.println(stack.pop()+
                       " "+queue.dequeue());
}
```

Queue Exceptions

```
public class QueueEmptyException
    extends RuntimeException
{
    public QueueEmptyException(String err){

    }
}
```

Queue Exceptions

```
public class QueueEmptyException
    extends RuntimeException
{
    public QueueEmptyException(String err){
        super(err);
    }
}
```

```
public class LinkedQueue<T>
implements Queue<T> {
private Node<T> head ;
private Node<T> tail ;
```

```
public class LinkedQueue<T>
implements Queue<T> {
private Node<T> head ;
private Node<T> tail ;
private int  length;
```

```
public class LinkedQueue<T>
implements Queue<T> {
private Node<T> head ;
private Node<T> tail ;
private int  length;

public LinkedQueue(){




}
```

```
public class LinkedQueue<T>
implements Queue<T> {
private Node<T> head ;
private Node<T> tail ;
private int  length;

public LinkedQueue(){
    head = null;
    tail = null;
    length = 0;
}
```

```
public int size() {

}
```

```
public int size() {
    return length;
}
```

```
public int size() {
    return length;
}
public boolean isEmpty() {

}
```

```
public int size() {
    return length;
}
public boolean isEmpty() {
    return (length == 0);
}
```

```
public T front()
    throws QueueEmptyException{




}
```

```
public T front()
    throws QueueEmptyException{
    if (isEmpty())
        throw new QueueEmptyException(
              "FRONT from empty queue");

}
```

```
public T front()
    throws QueueEmptyException{
    if (isEmpty())
        throw new QueueEmptyException(
              "FRONT from empty queue");
    return head.getData();
}
```

```
public void enqueue (T data) {
    Node<T> newNode = new Node<T>(data);









    length++;
}
```

```
public void enqueue (T data) {
    Node<T> newNode = new Node<T>(data);

    if (isEmpty())
        head = tail = newNode;
    else {



    }
    length++;
}
```

```
public void enqueue (T data) {
    Node<T> newNode = new Node<T>(data);

    if (isEmpty())
        head = tail = newNode;
    else {
        tail.setNext(newNode);
        tail = newNode;
    }
    length++;
}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");




}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");
    T data = head.getData();




}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");
    T data = head.getData();
    head = head.getNext();
    length--;

}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");
    T data = head.getData();
    head = head.getNext();
    length--;
    return data;
}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");
    T data = head.getData();
    head = head.getNext();
    length--;
    return data;
}
public void flush() {


}
```

```
public T dequeue()
    throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException(
              "DEQUEUE from empty queue");
    T data = head.getData();
    head = head.getNext();
    length--;
    return data;
}
public void flush() {
    while (!isEmpty())
        dequeue();
}
```

```
public void print() {
    Node p = head;
    while (p!=null) {




    }
    System.out.println();
}
```

```
public void print() {
    Node p = head;
    while (p!=null) {
        if (p !=head)
            System.out.print(", ");


    }
    System.out.println();
}
```

```
public void print() {
    Node p = head;
    while (p!=null) {
        if (p !=head)
            System.out.print(", ");
        System.out.print(p.getData());

    }
    System.out.println();
}
```

```
public void print() {
    Node p = head;
    while (p!=null) {
        if (p !=head)
            System.out.print(", ");
        System.out.print(p.getData());
        p = p.getNext();
    }
    System.out.println();
}
```

```
public void concatenate(LinkedQueue<T> q) {
    if (isEmpty()) {



    }
    else if (!q.isEmpty()) {



    }
}
```

```
public void concatenate(LinkedQueue<T> q) {
    if (isEmpty()) {
        head = q.head;
        tail = q.tail;
        length = q.length;
    }
    else if (!q.isEmpty()) {



    }
}
```

```
public void concatenate(LinkedQueue<T> q) {
    if (isEmpty()) {
        head = q.head;
        tail = q.tail;
        length = q.length;
    }
    else if (!q.isEmpty()) {
        tail.setNext(q.head);
        tail = q.tail;
        length += q.length;
    }
}
```

Using extra features of LinkedQueue

```
while(words.hasNext()){
    String word=words.next();


}
```

Using extra features of LinkedQueue

```
while(words.hasNext()){
    String word=words.next();
    queue1.enqueue(word);
    queue2.enqueue(word+2);
}
```

Using extra features of LinkedQueue

```
queue1.print();
queue2.print();
```

Using extra features of LinkedQueue

```
queue1.print();
queue2.print();
queue1.concatenate(queue2);
```

Using extra features of LinkedQueue

```
queue1.print();
queue2.print();
queue1.concatenate(queue2);
queue1.print();
queue2.print();
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
todo.enqueue(new Coord(startr, startc));
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
todo.enqueue(new Coord(startr, startc));
while(!todo.isEmpty()){









}
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
todo.enqueue(new Coord(startr, startc));
while(!todo.isEmpty()){
    Coord here = todo.dequeue();
    int r = here.r; int c = here.c;









}
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
todo.enqueue(new Coord(startr, startc));
while(!todo.isEmpty()){
    Coord here = todo.dequeue();
    int r = here.r; int c = here.c;
    int length = distance[r][c];









}
```

Using Queues to remember things, in order

QueueMaze

```
todo = new LinkedQueue<Coord>();
distance[startr][startc]=0;
todo.enqueue(new Coord(startr, startc));
while(!todo.isEmpty()){
    Coord here = todo.dequeue();
    int r = here.r; int c = here.c;
    int length = distance[r][c];
    visit(r-1,c,length+1);
    visit(r+1,c,length+1);
    visit(r,c-1,length+1);
    visit(r,c+1,length+1);
}
```

Using Queues to remember things, in order

```
private void visit(int i, int j,int length){
    if (i<0 || i>n || j<0 || j>m)
        return;




}
```

Using Queues to remember things, in order

```
private void visit(int i, int j,int length){
    if (i<0 || i>n || j<0 || j>m)
        return;
    if (map[i][j]==Cell.WALL)
        return;




}
```

Using Queues to remember things, in order

```
private void visit(int i, int j,int length){
    if (i<0 || i>n || j<0 || j>m)
        return;
    if (map[i][j]==Cell.WALL)
        return;
    if (distance[i][j]==Integer.MAX_VALUE){
        distance[i][j]=length;
        todo.enqueue(new Coord(i,j));
    }
}
```

Why do we use Queues for distances?

```
private LinkedStack<Coord> todo;
```

```
public void explore(){
```

Why do we use Queues for distances?

```
private LinkedStack<Coord> todo;
```

```
public void explore(){
    todo = new LinkedStack<Coord>();

    distance[startr][startc]=0;

    todo.push(new Coord(startr, startc));
```

Why do we use Queues for distances?

```
private LinkedStack<Coord> todo;
```

```
public void explore(){
    todo = new LinkedStack<Coord>();

    distance[startr][startc]=0;

    todo.push(new Coord(startr, startc));
    while(!todo.isEmpty()){
        Coord here = todo.pop();

    ⋮
    }
```

Why do we use Queues for distances?

StackMaze

```
private void visit(int i, int j,int length){
    ⋮
    if (distance[i][j]==Integer.MAX_VALUE){
        distance[i][j]=length;
        todo.push(new Coord(i,j));
    }
}
```