Lists, Stacks and Queues

Lists

The most basic type of data collection – a List ADT is a dynamic linear data structure. The data items would be accessible one after another starting from the head of the list.

The main operations are:

- Add data
- Remove data
- Query data

The full list of operations (List Interface) are:

```java
public interface ListInterface {
  public boolean isEmpty();
  public int size();
  public int indexOf(int item);
  public boolean contains(int item);
  public int getItemAtIndex(int index);
  public int getFirst();
  public int getLast();
  public void addAtIndex(int index, int item);
  public void addFront(int item);
  public void addBack(int item);
  public int removeAtIndex(int index);
  public int removeFront();
  public int removeBack();
  public void print();
}
```

List Implementation #1: Arrays
Can be done using Arrays. Refer to the section on Arrays to see how the operations are performed.

List Implementation #2: LinkedList
Can be done using LinkedList. Refer to the section on LinkedList to see how the operations are performed.

Which implementation is better?

Key Benefit of Arrays:
Access in O(1)
Less Overhead than LinkedList (no need for pointers)
Can reduce the high cost of enlarging by creating an ArrayList with a larger initial capacity (as default initial capacity is around 10.

Key Benefit of LinkedList:
Deletion and Insertion takes O(1) **if you have a reference to the node where the operation needs to be done.** This includes the tail and head (array is O(n) for head)

Less unused memory allocated, but the overhead > memory saved

Stacks

A stack is a collection of data that is accessed in a Last-In-First-Out (LIFO) manner.

The main operations are:

- Push
- Pop
- Peek

The full list of operations (Stack Interface) is:

```
public interface StackADT {
  // check whether stack is empty
  public boolean empty();

  // retrieve topmost item on stack
  public Integer peek(); // returns obj ver. of int

  // remove and return topmost item on stack
  public Integer pop(); // returns obj ver. of int

  // insert item onto stack
  public void push(Integer item);
}
```

Stack Implementation #1: Arrays

Use an Array with a top index pointer, which starts at -1 for an empty stack.

- isEmpty() – return top<0
- peek() – if isEmpty() return null else return arr[top]
- push(item) – if stack is full enlarge array else top++ and arr[top] = item
- pop() – if isEmpty() return null else return peek() and top—
- enlargeArr() – same as for Arrays

Stack Implementation #2: LinkedList

Use a LinkedList with the top of the stack being the head of the LinkedList

- isEmpty() – return linkedList.isEmpty()
- peek() – if isEmpty() return null else return head of linked list
- push() – add to front of linked list
- pop() – item = peek(), if not empty, remove first element of linked list

Which implementation is better?

LinkedList is generally better in terms of worst-case performance but will take more space. They have better worst-case behavior but may have worse overall runtime due to the large number of allocations required. Arrays have worse worst-case performance but overall better performance.

Analysis of Arrays:

Popping will always be O(1)
It takes less space than LinkedList as there is no need to store pointers
However, pushing may take up to O(n) time if it ends up enlarging the array. Else, it will also be O(1).

Analysis of LinkedList:
Pushing and popping will all be O(1)
Allocation of space can be expensive due to the need to maintain pointers

Applications:

- Calling a function
    - When another function is called within a function, the state of the caller function is saved on the stack, so we know where to resume
- Recursion using the above logic
- Matching parentheses
    - For every bracket in the expression
        - If it's an open bracket, push it into the stack
        - If it's a closed bracket, pop a bracket out of the stack
            - If the brackets do not match, throw error
    - If the stack is not empty after evaluating the expression, throw error
- Evaluating arithmetic expressions
    - Postfix calculation
        - Create an empty stack
        - For each item of the expression
            - If it is an operand
                - Push it on the stack
            - If it is an operator
                - Pop arguments from stack
                - Perform the operation
                - Push the result onto the stack
        - Pop the final result out
    - Infix to postfix conversion
- Traversing a maze together with a graph

Queues
A queue is a collection of data that is access in a First-In-First-Out (FIFO) manner.

The main operations are:

- Offer/Queue
- Poll/Dequeue
- Peek

The full list of operations (Queue interface) is:

```
public interface QueueADT {
  // return true if queue has no elements
  public boolean empty();

  // return the front of the queue
  public Integer peek();

  // remove and return the front of the queue
  public Integer poll(); // also known as dequeue

  // add item to the back of the queue
  public void offer(Integer item); // also known as enqueue
}
```

Queue Implementation #1: Circular Arrays
Using an Array with a front index and a back index, we can implement a Circular Array to recycle space, with a gap between the front and back index.

- isEmpty() – return front==back
- offer(item) – if array is full, enlargeArr, else arr[back] = item, and back = (back+1)%maxSize
- poll() – if isEmpty() return null else item = arr[front] and front = (front+1)%maxSize, return item
- peek() – if isEmpty() return null else return arr[front]
- enlargeArr() – same as Arrays except we loop using j=0; i<maxSize; j++ and arr[(front+j)%maxSize]

Queue Implementation #2: Tailed Linked Lists
Tailed Linked List needs to be used as we need to use the addBack() method.

- isEmpty() – return list.isEmpty()
- offer(item) – list.addBack(item)
- poll() – item = peek(), if not isEmpty(), then list.removeFront() before returning the item
- peek() – if isEmpty(), then return null, else return list.getFirst()

Which implementation is better?
Similar to a Stack, a Linked List implementation has better worst case performance, but may have poorer runtime due to the number of allocations required.

Analysis of Arrays:

Dequeuing takes O(1) time. Queuing takes O(1) time, unless the array is full and enlarging is required. In that case, the time complexity becomes O(n).

Analysis of LinkedList:
Both queuing and dequeuing takes O(1) time, but time may be required for a new allocation for every enqueue.

Applications:
A queue is useful when we are trying to queue things that we may not be processing immediately. An example would be breadth first search for graphs.

Application of Stack and Queue Combined:
Stack and queue together can be used to identify palindromes. This can be done by pushing a string into both a queue and stack by character, then popping them out to compare.

Deque

A double-ended queue is a queue where you can enqueue and dequeue from both ends of the queue. In other words, it can somewhat act as a Stack and a Queue at the same time, although its main purpose is to allow people to push and pull from both ends conveniently.

Implementation:
Both Circular Arrays and Tailed LinkedList can be used. So long a reference can be maintained at the head and the tail, the addition and removal of elements can be easily done from both ends.

Implementation using Arrays:
Simply maintain a front and back index, with an additional capability of adding to the front and removing from the front.

- addFront(item) – if not isEmpty(), front = (front+maxSize-1)%maxSize, arr[front] = item, else if isEmpty(), arr[front] = item.
- removeFront() – if isEmpty() return null, else item = arr[front], front = (front+1)%maxSize, return item.

Implementation using LinkedList:
Will be the exact same as using a Tailed LinkedList, with the addFront(), addBack(), removeFront(), removeBack() functions.

Which implementation is better?
Generally, using an Array for a Deque is much better, as it has O(1) getting due to random access memory, and due to the cache locality. The only issue is the need to enlargeArray should the size be insufficient.

LinkedList may encounter a cache miss for every element, and a lot more memory is needed to maintain the next and previous pointers for each node.