Linear Structures

Agenda

1. Arrays
2. Linked Lists
3. Queues
4. Stacks
5. Applications

Arrays

Array

- Arrays are objects of a “built-in” class in Java, which can be used to store a fixed number of elements of the same type.
- There are other types of “arrays” implemented in packages
- Elements in an array can be accessed via indices, ranging from 0 to (array.length – 1).

Array

In Java - an array is formed via 3 steps: declare, create, and initialise.

```
// declare -> no memory allocation at this point
int[] a;
// create -> allocate memory
a = new int[5];
// initialize ; only assign one element at a time
a[0] = 1;
…
// another way: int[] a = {1, 2, 3, 4, 5};
```

Array

- Without initialisation, array elements are set to default values
  - `0` for numeric data types
  - `\u0000` for `char` types
  - `false` for `Boolean` types
- If you want to change the number of elements in an array, the only way is to create a new array then clone the data to the new array

Multi-dimensional Array

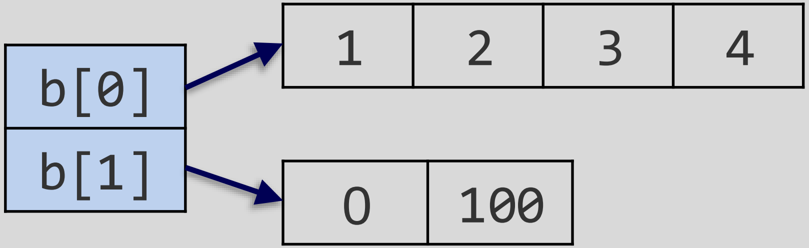
You can use a multi-dimensional array to store a matrix or a table (first brackets are rows; second brackets are columns)

```
double[][] distances = {{0, 2, 5, 1},
                        {2, 0, 3, 2},
                        {5, 3, 0, 1},
                        {1, 2, 1, 0}}

distances[0][2] = 5;

distances[3][1] = 2;
```

Multi-dimensional Array

- Elemental arrays can have different lengths, but must be of the same type.

```
int[][] b = {
              {1,2,3,4},
              {0,100}
            };
```



Multi-dimensional Array

- Similarly, n-dimensional arrays can be created:

```
// create integer array of size 2*3*4
int[][][] a = new [2][3][4];
// rows = 2, columns = 3, cells = 4
```

{0,0,0,0}	{0,0,0,1}	{0,0,1,0}
{0,1,0,0}	{1,0,0,0}	{1,1,1,1}

Multi-dimensional Array

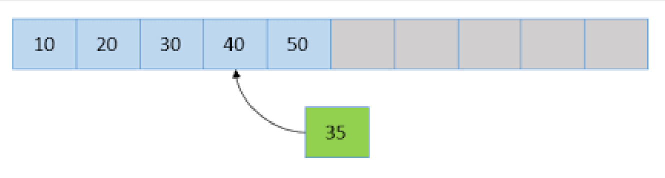
```
// create integer array of size 2*3*4
int[][][] a = {
                {{0,0,0,0}, {0,0,0,1}, {0,0,1,0}},
                {{0,1,0,0}, {1,0,0,0}, {1,1,1,1}}
                };
```

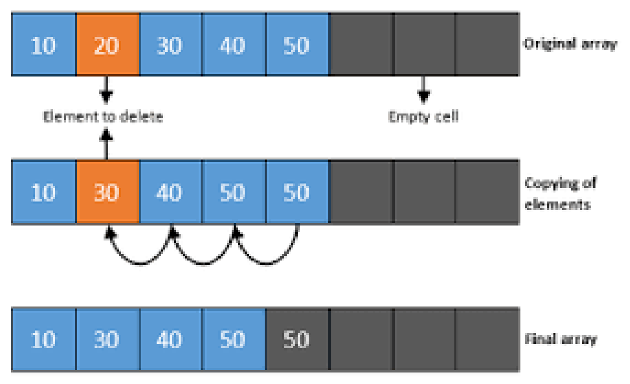
{0,0,0,0}	{0,0,0,1}	{0,0,1,0}
{0,1,0,0}	{1,0,0,0}	{1,1,1,1}

```
// 1st row, 2nd column, 1st element
a[0][1][0] = 0;
```

Issues with Arrays

- Need to pre-define the maximum size – a lot of wasted space
  - Some languages do have the concept of dynamic arrays
- How do we add or delete items….
  - Adding – can easily add new item to the end of the array - but if the array is sorted we need to shift all the items to the right
  - Deleting – we need to shift all the items to the left or have to manage sparse arrays
- Question – where do we have to manage sparse arrays?





Array – Time & Space Complexity

ARRAY	Average	Worst
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)
Space	O(n)	O(n)

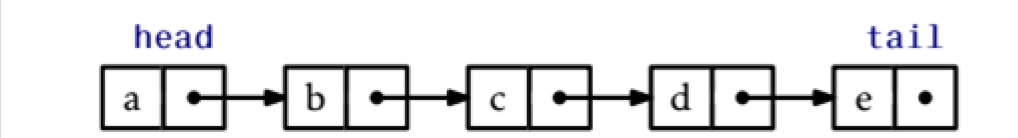
Array – Performance Analysis

- Strengths
  - Convenient for storing and processing a collection of data
  - Randomly access in constant time
- Weaknesses:
  - Fixed size → have to determine array size upfront
  - Inserting or removing elements are costly
- We need another data structure that
  - Is faster to insert/remove
  - Can be expanded/shrunk easily

Singly Linked Lists

Linked List

- Linked list is a sequence of nodes; each node has connection(s) to the node(s) next to it.
- What is a node? Think about a cabin in a train
  - Space to contain data (the cabin)
  - Reference(s) to the next node(s) (the hooks)
- Add/insert/remove elements → just update the hooks



Linked List

The first node in the sequence is the head, the last node in the sequence is the tail.

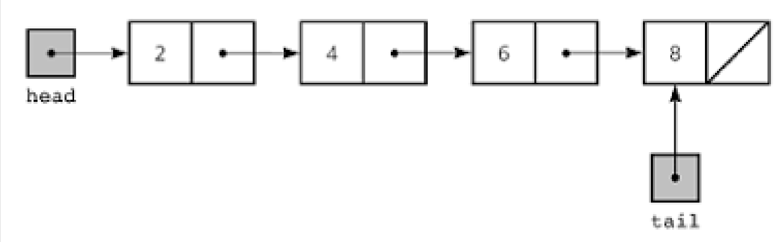
A Node is used to wrap both the data and the pointer to the next Node.

A List is a collection of Node.

```
static class Node<T> {
    T data;
    Node<T> next;

    public Node(T data) {
        this.data = data;
        this.next = null;
    }
}
```

SLL – Simple Implementation

```java
public class LinkedList<T> implements List<T> {
  private int size;
  private Node<T> pointer;
  private Node<T> head;

  public LinkedList() {
    size = 0;
    head = null;
    pointer = null;
  }

  @Override
  public int size() {
    return size;
  }
```



- The size attribute stores the number of elements in a list.
- The pointer attribute is used to iterate through the list elements.

SLL – Insert

- Insert at the head

```java
private boolean insertAtHead(T value) {
    Node<T> n = new Node<T>(value);
    n.next = head;
    head = n;
    size++;
    return true;
}
```

- Insert at any location

```java
@Override
public boolean insertAt(int index, T value) {
    if (index > size) {
        return false;
    }
    if (index == 0) {
        return insertAtHead(value);
    }
    Node<T> current = head;
    Node<T> previous = null;
    while (index > 0) {
        previous = current;
        current = current.next;
        index--;
    }
    Node<T> node = new Node<T>(value);
    node.next = current;
    previous.next = node;
    size++;
    return true;
}
```

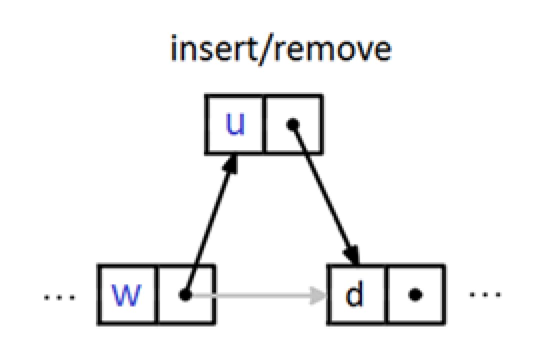
SLL – Insert Before/After a Node

- Insert before/after a Node

```java
@Override
public boolean insertBefore(T searchValue, T value) {
  if (head == null) {
    return false;
  }
  if (head.data.equals(searchValue)) {
    return insertAtHead(value);
  }
  Node<T> current = head;
  Node<T> previous = null;
  while (current != null) {
    if (current.data.equals(searchValue)) {
      Node<T> node = new Node<T>(value);
      node.next = current;
      previous.next = node;
      size++;
      return true;
    }
    previous = current;
    current = current.next;
  }
  return false;
}
```

```java
@Override
public boolean insertAfter(T searchValue, T value) {
  if (head == null) {
    return false;
  }
  Node<T> current = head;
  Node<T> previous = null;
  while (current != null) {
    if (current.data.equals(searchValue)) {
      previous = current;
      current = current.next;
      Node<T> node = new Node<T>(value);
      node.next = current;
      previous.next = node;
      size++;
      return true;
    }
    previous = current;
    current = current.next;
  }
  return false;
}
```

SLL – Remove ith Node




```java
private boolean removeAtHead() {
    if (head == null) {
        return false;
    }
    head = head.next;
    size--;
    return true;
}
```

```java
@Override
public boolean removeAt(int index) {
    if (index >= size) {
        return false;
    }
    if (index == 0) {
        return removeAtHead();
    }
    Node<T> current = head;
    while (--index > 0) {
        current = current.next;
    }
    current.next = current.next.next;
    size--;
    return true;
}
```

SSL – Search

- Find i^{th} node: starting from head node, then traverse the list to reach the node of interest → it is O(n).

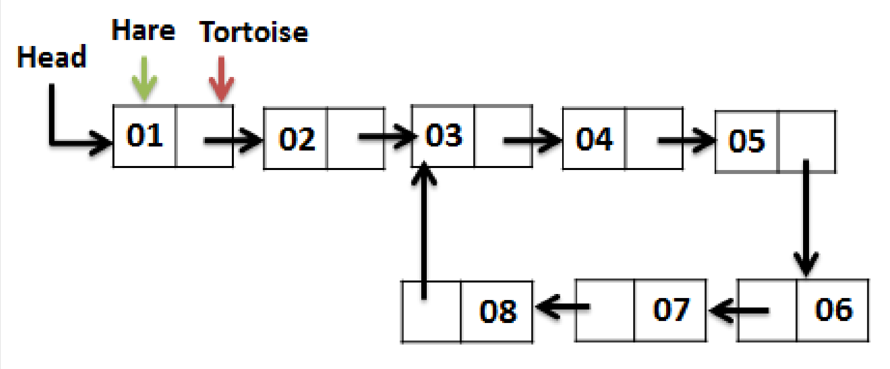
```java
@Override
public T get(int index) {
  if (index >= size) {
    return null;
  }
  Node<T> p = head;
  while (index > 0) {
    p = p.next;
    index--;
  }
  return p.data;
}
```

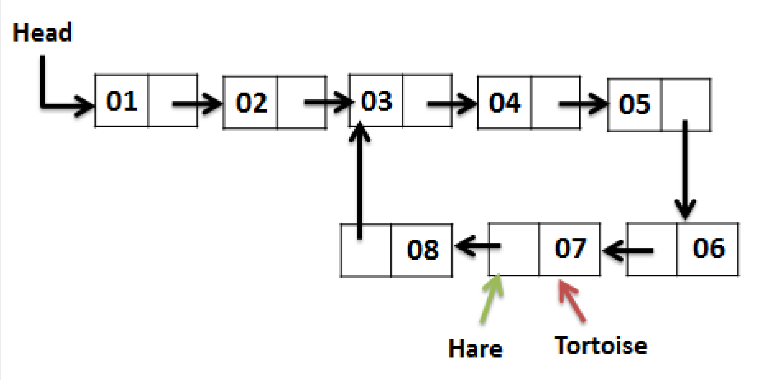
Cannot traverse in backward direction

Discussion - Using a Linked List

- How do you delete an element? What complexities do we need to address?
- How do you find an element in the list? What issues do we need to address?
- How do you find the last element in the list?
- How do you find the 3rd element from the end of a linked list?
- How would you find a loop in a linked list?

Floyd's Algorithm





The pattern of Hare and Tortoise movements are shown below.

Hare	Tortoise
1	1
3	2
5	3
7	4
3	5
5	6
7	7

Complexity:
Time Complexity: **O(n)**
Space Complexity: **O(1)**

Floyd's Cycle Detection Algorithm

Aka as the “Tortoise and Hare Algorithm"

1. Start Tortoise and Hare at the first node of the List
2. If Hare reaches end of the List, return as there is no loop in the list
3. Else move Hare one step forward
4. If Hare reaches end of the List, return as there is no loop in the list
5. Else move Hare and Tortoise one step forward
6. If Hare and Tortoise point to same Node - return, found loop in the List
7. Else start with STEP 2

How would you solve these?

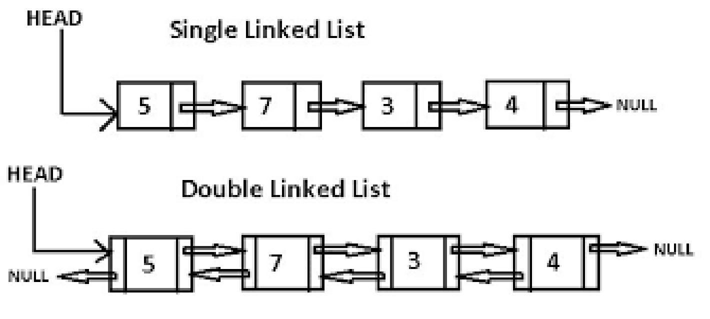
- Print every second entry in a linked list
- Print/delete the middle element of a linked list
- Removing the duplicates in a linked list?
- How do you swap every two nodes in a linked list without moving the data: For example 1, 2, 3, 4, 5, 6 becomes 2, 1, 4, 3, 6, 5

- Note: you can use only arrays/linked lists to solve above tasks

Doubly Linked Lists

Doubly Linked List (DLL)

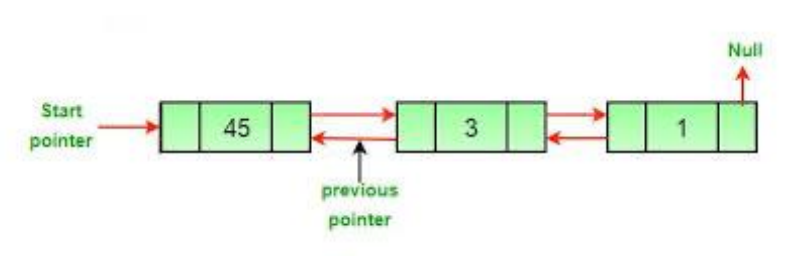
- DLL is very similar to SLL, however each node in DLL (except head and tail) has references to ***both*** the node follows it and the node precedes it.




```
class DNode {
    // data of generic type T
    T x;
    // reference to adjacent DNodes
    DNode prev, next;
}
```

Doubly Linked List (DLL)

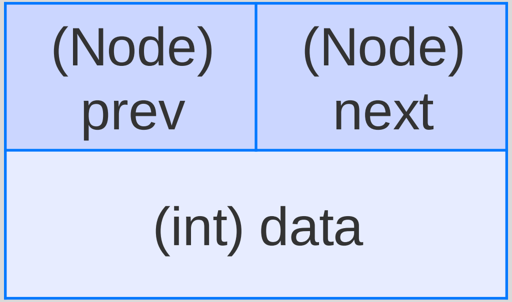
- In DLL, **head.prev** and **tail.next** are null



- Alternatively, a dummy node may be created to be referenced from and to the tail and the head respectively

DLL – Simple Implementation

```java
// Implement the Node class
public class Node {
     int data;
     Node prev, next;

     public Node(int value) {
         data = value;
         prev = null;
         next = null;
     }
}
```



DLL – Simple Implementation

```
// Implement the Doubly Linked List class
public class DLL {
    Node head, tail;
    int length;

    public DLL(int value) {   // create a list of 1 node
        head = new Node(value);
        tail = head;
        length = 1;
    }
}
```

DLL Append and Print

```
public void appendNode(Node aNode) {
    tail.next = aNode;    // at the end of the list
    aNode.prev = tail;
    tail = tail.next;
    length += 1;                 // keep track of size
}
public void printList() {
    Node cur = head;
    while (cur != null) {
        System.out.println(cur.data);
        current = cur.next;   // traverse the list
    }
}
```

DLL – Search

- Starting either from head node or tail node, then **traverse** the list to reach the node of interest
- $O(1 + \min\{i, n - i\})$ to reach i^{th} node (twice faster than in SLL)
- However, how you know from which end to start??

DLL – Search

```
Node getNode(int i) {
    Node p = null;
    if (i < n / 2) {
        p = head;
        for (int j = 0; j < i; j++)
            p = p.next;
    } else {
        p = tail;
        for (int j = n-1; j > i; j--)
            p = p.prev;
    }
    return p;
}
```

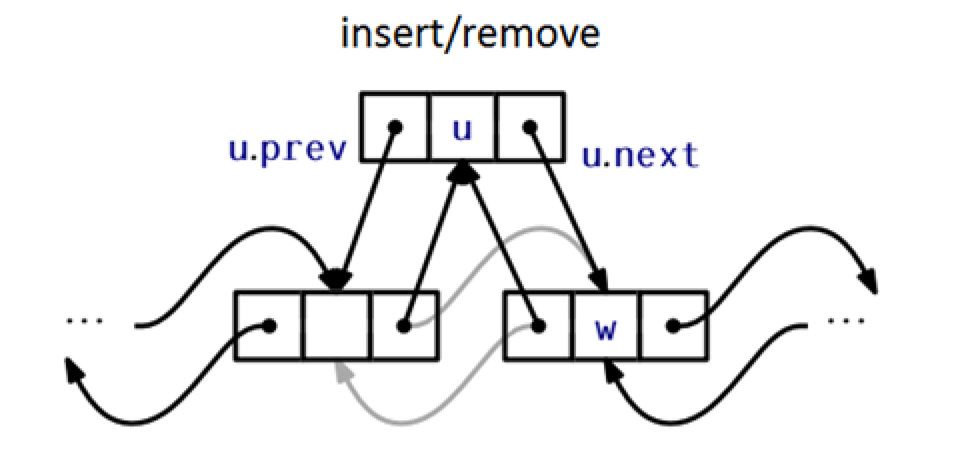
Does that mean DLL is better than SLL?

→ There are twice number of references to handle.

Get and set value of i^{th} node: similar to SLL

DLL – Insert before node w

```
Node addBefore(Node w, T data) {
    Node u = new Node();
    u.data = data;

    u.prev = w.prev;
    u.next = w;

    u.next.prev = u;
    u.prev.next = u;

    length++;
    return u;
}
```



LL – Time & Space Complexity

Linked List	Average SLL / DLL	Worst SLL / DLL
Access	O(n) / O(n)	O(n) / O(n)
Search	O(n) / O(n)	O(n) / O(n)
Insertion	O(1) / O(1)	O(1) / O(1)
Deletion	O(1) / O(1)	O(1) / O(1)
Space		O(n) / O(n)

Sorted vs Unsorted Lists

- Searching an unsorted list
  - Compare the search item with the current node in the list. If the info is the same stop the search; otherwise, make the next node the current node
  - Repeat until either the item is found - or no more data is left in the list
- Searching a sorted list
  - Compare the search item with the current node in the list. If the info of current node is greater or equal than the search item, stop; otherwise, make the next node the current node
  - Repeat until either an item in the list that is greater than or equal to the search item is found - or no more data is left in the list
- What is the complexity of each approach?

Linked List Discussion

- Searching sorted and unsorted lists is O(n) - but, on average, sorting lists are twice as fast.
- Inserting into unsorted lists is O(1) and to sorted lists is O(n).
- LL are more complex to implement and use than an array….
- Not all data collections are sequential: collection of books, structure of a company, etc.
- No specific logic on the sequence of the items in the list
- A big part of node is not used to store data
- O(n) complexity limits to find and retrieve limits their usefulness

Comparing Arrays vs List

- Arrays easy to use, but have fixed size - linked lists are variable size
- Next Item in an array is *implied.* Next item in linked list is *explicit*
- Array based implementations requires less memory per item
- Array items accessed directly and have equal access time - O(1). One must traverse a linked list for i^{th} item – access time varies - O(n).
- Static array-based implementations are good for small number of items. Linked lists are best when the number of items can be large
- Using dynamically allocated arrays will waste storage and time. Link based implementations are exactly as long as the number of items

Special Lists

Array-based List

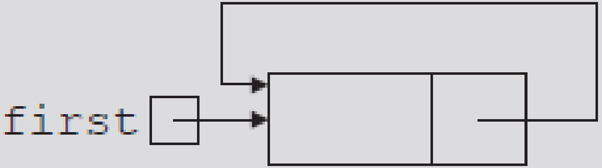
- Depend on the application, either the ***constant time access*** or the ***changeable size*** gets favour
- If access time is more important, a list can be built on a array
- Array indices play role of references
- An array larger than the list size is normally used to reserve some space
- If the list size grows exceeded array size, then new array is created and data is transfer.

Circular List

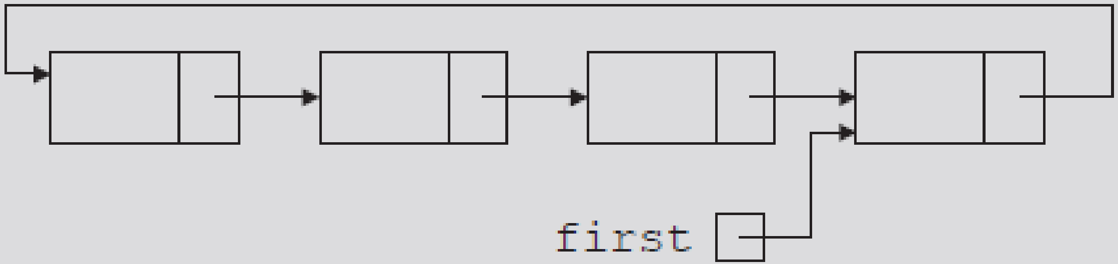
- In both SLL and DLL, we have to keep track of **head** and **tail** references.
- Mistakes in updating these two references may result in data lost or program failure
- One way to avoid these reference issues is to use a **circular DLL**.
- In this structure, the **tail** and **head** are connected to each other to make a circle, and a **dummy node** (contains no data) is used as a starting point of the list.

Circular List

(a) Empty circular list



(b) Circular linked list with one node

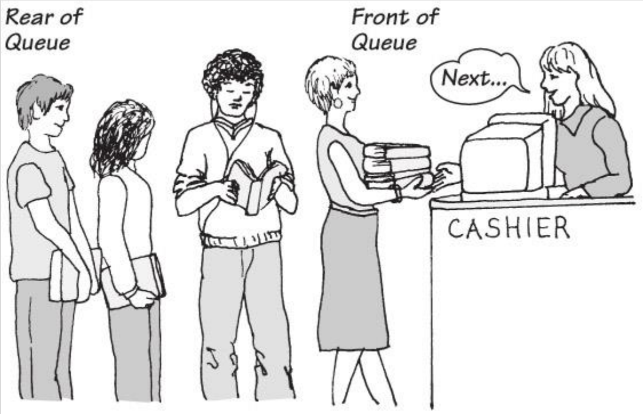


(c) Circular linked list with more than one node

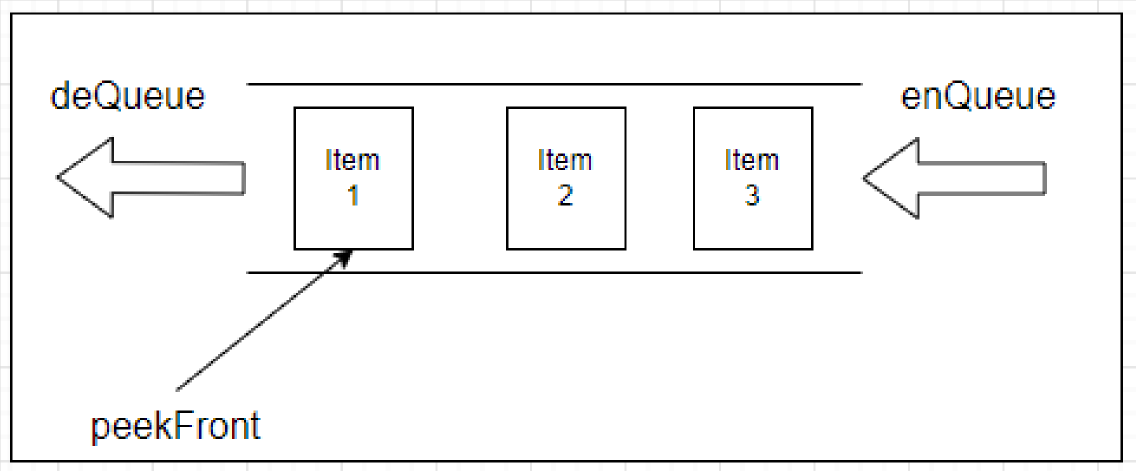
Queues

The Queue ADT

- A queue is like a line of people – FIFO (First in, first out)
  - New items enter at the back (rear) of the queue
  - Items leave the queue from the front
- Example
  - Teller at a bank or cashier in a supermarket
- Queue operations include:
  - Test whether a queue is empty
  - Add new entry to back of queue
  - Remove entry at front of queue
  - Read entry at front of queue



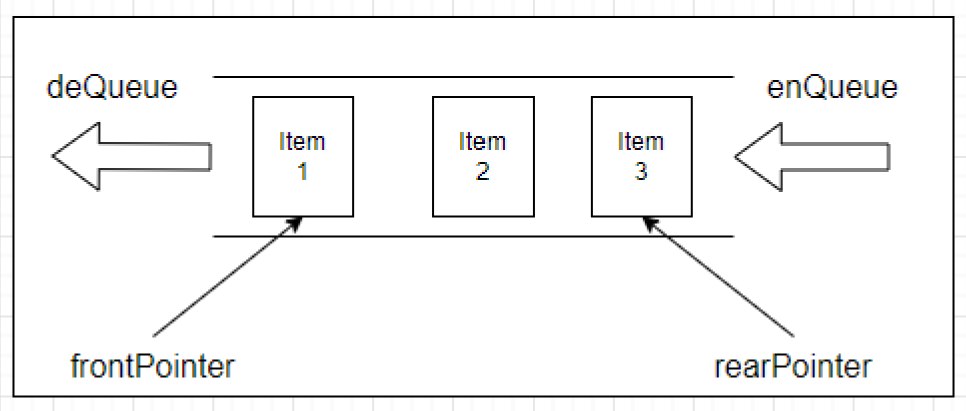
Queue Operations



- enQueue
- deQueue
- peekFront

Queue Implementation

- Using an array as the underlying data structure



- Create an array arr of size N (the maximum number of elements in the queue)
- Maintain two pointers: front and rear

Array–Based Queue Implementation - 1

- front is always zero
- enQueue
  - arr[rear] = newElement
  - rear++
- deQueue
  - rear--
  - for i is from 0 to (rear – 1) // shift all elements to the left
    - arr[i] = arr[i+1]
- peekFront
  - return arr[0]
- The complexities of enQueue, deQueue, peekFront operations?

Array–Based Queue Implementation - 2

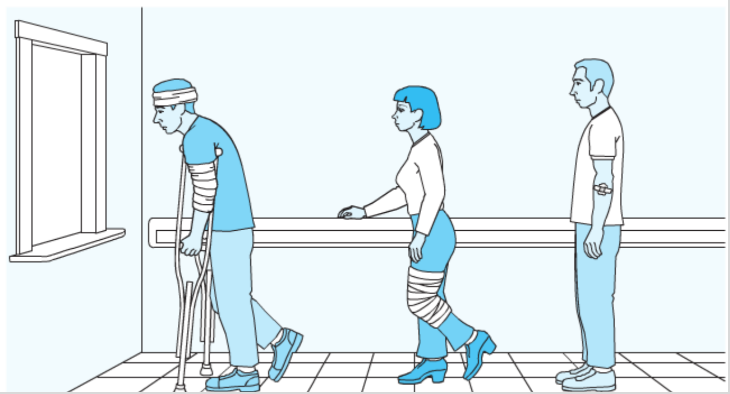
- Use both front and rear
- enQueue
  - arr[rear] = newElement
  - rear = (rear + 1) % N
- deQueue
  - front = (front + 1) % N
- peekFront
  - return arr[front]
- How to check if a queue is full or empty?
- What are the complexities in this case?

List–Based Queue Implementation

- Use a linked list instead of an array
- Change from front to head and rear to tail
- enQueue: append a new node to the tail
- deQueue: remove the node pointed to by head
- peekFront: return the node pointed to by head
- What are the complexities?

The Priority Queue

- Operations
  - Test whether priority queue empty
  - Add new entry to priority queue in sorted position based on priority value
  - Remove from priority queue entry with highest priority
  - Get entry in priority queue with highest priority
- Example
  - Hospital emergency room



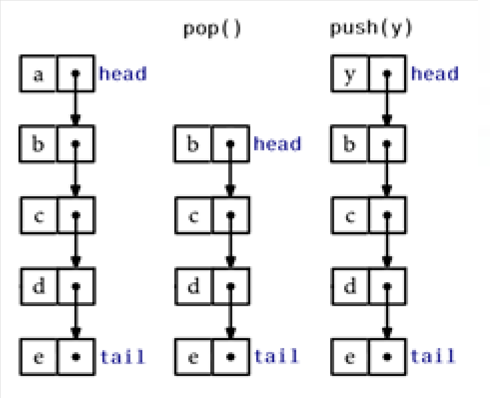
Priority Queue Implementation

- Below is a naïve approach for the enQueue operation
  - Find the correct position of the new element
    - How?
  - Insert the element at the correct position
    - If an array is used as the underlying data structure, a right shift is required before doing the insertion
- What is the complexity of this enQueue?
- Can we do better?
  - Use the heap data structure (learn later)

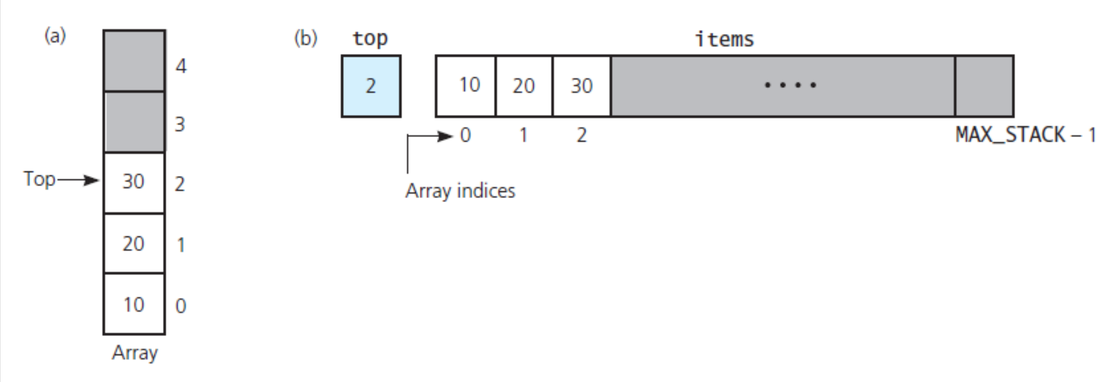
Stacks

The Stack ADT

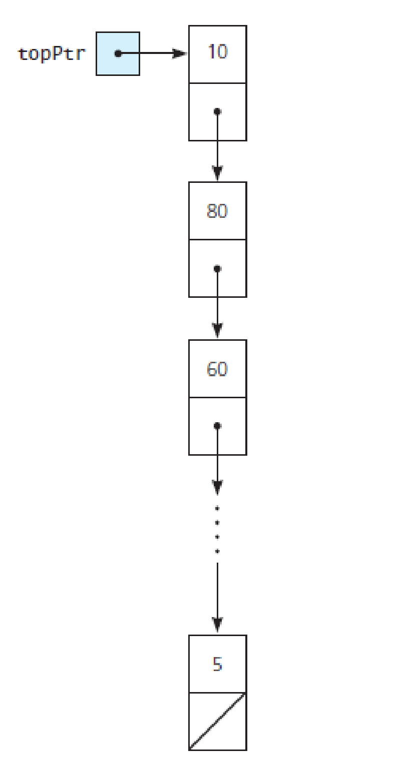
- A stack is like a tower of building blocks – LIFO (Last in, first out)
    - New items enter the top of the stack
    - Items leave the stack from the top
- Stack operations
    - Test whether a stack is empty
    - Push an entry to top of the stack
    - Pop the top of the stack
    - Read the last entry added to the stack



Stack Implementations



Array Based Implementation



Link Based Implementation

Application - Matching Parenthesis

- An expression can contain 3 types of delimiters (). {}, []
  - Valid - abc[d(ef)gh]i{jk}
  - Invalid - abc[d(ef)gh}i{j], abc(de}
- Use a stack to solve this task

For each char in string

If char = {, (, or [then stack.push {, (or [respectively

If char = },) or] then

if stack.peek == a {, (or [respectively then pop else fail

If stack is empty then succeed else fail

Application - Infix Expressions

- Grammar that defines language of fully parenthesized infix expression

$$\begin{aligned} <infix> &= <identifier> \,|\, (<infix> <operator> <infix>) \\ <operator> &= + \,|\, - \,|\, * \,|\, / \\ <identifier> &= a \,|\, b \,|\, \ldots \,|\, z \end{aligned}$$

- What are the values of the following expressions?
  - 5+2+3, 5 - 2 * 2, 5 - 4 - 3, 5 - 4 + 3, 15/4/2, 2** 3**2, 8÷2*(4+4)
- Are there other/better ways of writing an arithmetic expression?

Algebraic Expressions

- Infix - binary operator appears between its operands: a + b
- Prefix - operator appears before its operands: + a b
- Postfix - operator appears after its operands: a b +
- Note that when converting, the sequence of the operands is the same, just the operators need move and parentheses are removed

- No need for ( ) in prefix/postfix
  - a+(b*c), + a * b c, a b c * +
  - (a + b) * c, * + a b c, a b + c *
- Convert to prefix and postfix
  - (a + b) * (c – d)
  - 2+3+4
  - (2+3) * 4
  - (2+3) * (5-4)
  - ((2 + (6*3)) – (5+2))

Evaluating Postfix Expressions

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

Postfix Expr Evaluation Algorithm

```
stack s = empty
for each postfix expression token T (from left to right)
    if T == operand
        s.push(T)
    else
        operand1 = s.peek()
        s.pop()
        operand2 = s.peek()
        s.pop()
        s.push(result of (operand2 T operand1))
return s.peek()
```

Converting Infix to Postfix

ch	aStack (bottom to top)	postfixExp
a		a
–	–	a
(	– (	a
b	– (	ab
+	– (+	ab
c	– (+	abc
*	– (+ *	abc
d	– (+ *	abcd
)	– (+	abcd*
	– (	abcd*+
	–	abcd*+
/	– /	abcd*+
e	– /	abcd*+e
		abcd*+e/–

Converting
a - (b + c * d) /e

Move operators from stack to postfixExp until "("

Copy operators from stack to postfixExp

Infix to Postfix Algorithm

```
stack s = empty, postfixExp = empty
while (infixExp != empty)
    token = extract next token from infixExp
    if token == operand
        add token to postfixExp
    else
        while (precedence(token) <= precedence(top(s)))
            pop(s) and add to postfixExp
        push token to s
while (s != empty)
    pop(s) and add to postfixExp
```

Comparison of Stack and Queue

- Can be implemented using an Array or a List
- Push inserts a new item at the top of the stack, enqueue inserts it at the rear of queue
- Pop removes the most recent item from the top of the stack, dequeue removes the first item from the front
- peek gets the most recent item from the top, peekFront gets the first item from the front of the queue
- isEmpty checks if any items exist in the ADT

Queue
+isEmpty(): boolean +enqueue(newEntry: ItemType): boolean +dequeue(): boolean +peekFront(): ItemType

Stack
+isEmpty(): boolean +push(newEntry: ItemType): boolean +pop(): boolean +peek(): ItemType

RMIT
UNIVERSITY