Data Structures

Topics Covered:

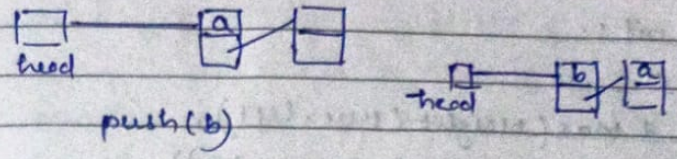
Stack

Queue

Tree

Dynamic Arrays

Stack implementation with linked list

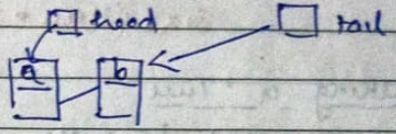


Queue:

Abstract data type with the following operation:

- Enqueue (key): add key to collection.
- Key Dequeue (): remove and return least recently-added key.
- Boolean Empty: are there elements?

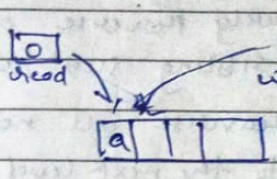
FIFO



Queue Implementation with Array:

enqueue(b)

while dequeue first a will remove



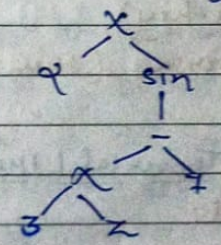
Summary:

- Queues can be implemented with either a linked list (with tail pointer) or an array
- Each queue operation is O(1): Enqueue, Dequeue, Empty

Trees: Syntax of sin(3x-7)

↓

Geography Hierarchy.



A Tree is:

- empty or,
- a node with:
  - a key and
  - a list of child trees.

```
Height(tree)
if tree = nil:
    return 0
return 1 + Max(Height(tree.left),
               Height(tree.right)).

Size(tree)
if tree = nil
    return 0
return 1 + size(tree.left) +
           Size(tree.right)
```

* Walking a Tree

Often we want to visit the node of a tree in a particular order.

- Depth-First Search: We completely traverse one sub-tree before exploring a sibling sub-tree
- Breadth-first search: We traverse all nodes at one level before progressing to the next level

* Depth-First

```
InOrderTraversal(tree)
if tree = nil:
    return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)

PreOrderTraversal(tree)
if tree = nil:
    return
Print(tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

PostOrderTraversal (tree)

```
if tree = nil:
    return
PostOrderTraversal (tree.left)
PostOrderTraversal (tree.right)
Print (tree.key)
```

Breadth-First:

- LevelTraversal (tree)

```
if tree = nil : return
Queue q
q.Enqueue (tree)
while not q.Empty():
    node ← q.Dequeue()
    Print (node)
    if node.left ≠ nil:
        q.Enqueue (node.left)
    if node.right ≠ nil:
        q.Enqueue (node.right)
```

Dynamic Arrays: Solve Problems if might not know size when allocating an array

↓

Abstract data type with the following operation (at a minimum)

- Get(i): returns element at location i*
- Set (i, val): Sets element i to val*
- PushBack(val): Adds val to the end
- Remove (i): Removes element at location i
- Size(): the number of elements

Get (i)

```
if i<0 or i ≥ size:
    return : index out of range
```

Set(i, val)

if i<0 or i ≥ size:
    ERROR : index out of range
arr[i] = val

PushBack (val)

if size = capacity:
    allocate new_arr[2 x capacity]
    for i from 0 to size - 1:
        new_arr[i] ← arr[i]
    free arr
    arr ← new_arr; capacity ← 2 x capacity
arr[size] ← val
size ← size + 1

Remove (i)

if i<0 or i ≥ size:
    ERROR : index out of range
for j from i to size - 2:
    arr[j] ← arr[j+1]
size ← size - 1

Size()
return size

Runtime:

Get (i)	O(1)
Set (i, val)	O(1)
PushBack (val)	O(n)
Remove	O(n)

Amortized Analysis - Aggregate Method.

Amortized cost: given a sequence of n operations, the amortized cost is: Cost (n operations) / n

Aggregate Method:
    Dynamic array: n calls to pushBack