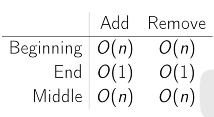
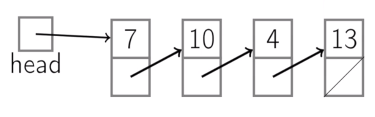
3/31/2021

Array:

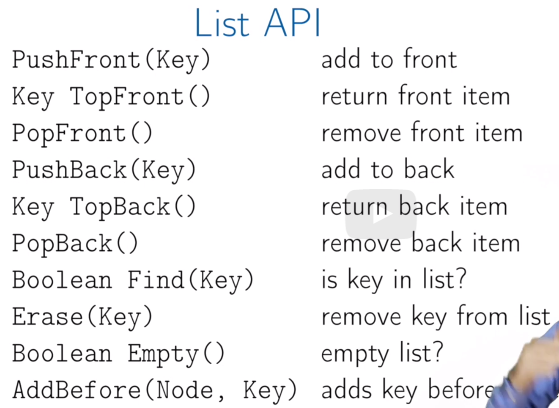
1. Array: contiguous memory of equal size elements indexed by contiguous integers
   1. constant-time access: arraddr + elemsize \*(i - first index)
2. Multi-dimensional Array: (3,4) in (3,6) = (3-1)\*6+(4-1)
3. row-major indexing: (1,1),(1,2),...,(2,1),...



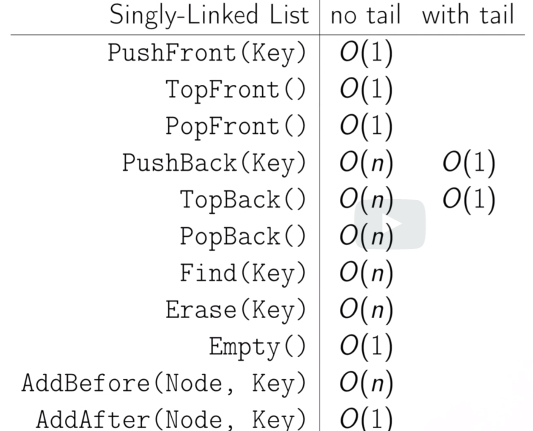
Linked List:

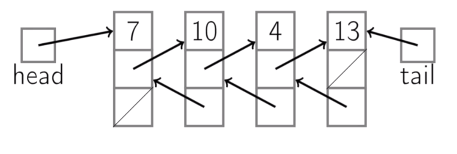


1. Node contains: key and next pointer
2. Operations



1. If we have a tail pointer
   1. Consider poptail O(n)



1. Doubly-linked List
   1. 

Stack:

1. Push, Top, Pop (all O(1))
2. Q: balanced brackets
3. Stack with array (array has maximum size)
4. Stack with linked list with tail
   1. Push at front
   2. Pop front

Queue:

1. Enqueue, Dequeue, Empty (all O(1))
2. Queue with linked list with tail
   1. enqueue at tail
   2. dequeue front
3. Queue with array
   1. Read, write pointers for next enqueue, dequeue position
   2. not write one element

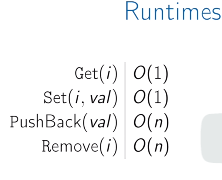
Tree:

1. Tree is:
   1. Empty
   2. A node with
      1. A key and
      2. A list of child trees
2. Root, child, parent, ancestor, descendent, sibling, leaf, interior node, level (1+number of edges between root and node (root level = 1)), height (maximum depth of subtree node and farthest leaf (leaf height = 1)), forest
3. binary search tree (BST)
   1. at most 2 children each node
   2. left <= root node <= right
   3. key, left, right, parent (optional)
4. recursive for height(tree), size(tree)

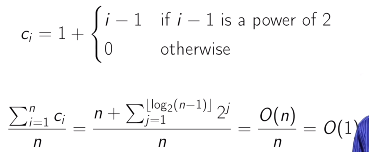
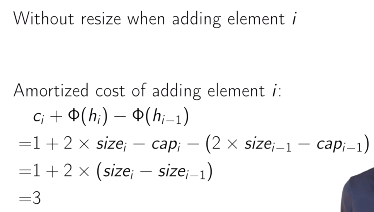
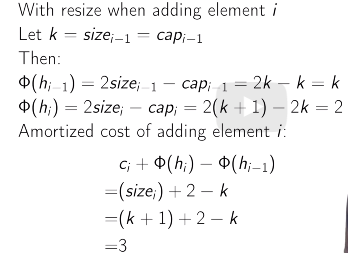
Tree Traversal:

1. Depth-first:
   1. Descendent first
   2. Recursive (InOrderTraversal, ascending, only for BST)
      1. Or use stack
      2. Visit left most leaf first
   3. PreOrderTraversal(tree)
      1. Visit node first and then children
   4. PostOrderTraversal
      1. Visit all leaves of subtree first
2. breadth-first
   1. sibling first
   2. LevelTraversal
      1. Use Queue

Dynamic Array:

1. Get(i), set(I, val), pushback, remove, size (first 2 O(1))
2. Arr, capacity, size
3. 

Amortized Analysis:

1. Most time O(1) sometime O(n)
2. Amortized cost = Cost(n operations)/n
3. N calls to pushback (aggregate method):
4. 
5. Banker’s method
   1. Charge extra for each cheap operations
   2. Save extra charge as tokens in data structure
   3. Use the tokens to pay for operations
   4. Charge 3 for each insertion:
      1. 1 for insertion
      2. 1 for itself moving to new array
      3. 1 for 1 prior moving to new array
6. Physicist’s method
   1. Define a potential function, which maps states of the data structure to integer
   2. Phi(h\_0) = 0 (initial state)
   3. Phi(h\_t) >= 0
   4. Amortized cost for operation t
      1. C\_t + phi(h\_t) - phi(h\_(t-1))
   5. Choose phi s.t. c\_t small, potential increase; c\_t large, potential decrease on the same scale
   6. Let phi(h) = 2 \* size – capacity
   7. 
   8. 
   9. 