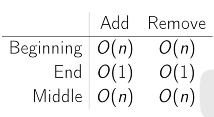
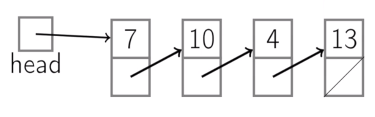
3/31/2021

Array:

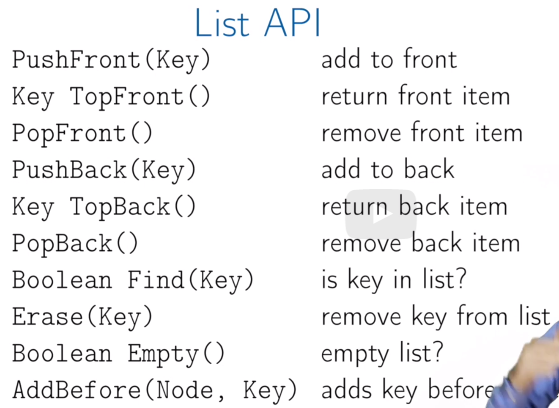
1. Array: contiguous memory of equal size elements indexed by contiguous integers
   1. constant-time access: arraddr + elemsize \*(i - first index)
2. Multi-dimensional Array: (3,4) in (3,6) = (3-1)\*6+(4-1)
3. row-major indexing: (1,1),(1,2),...,(2,1),...



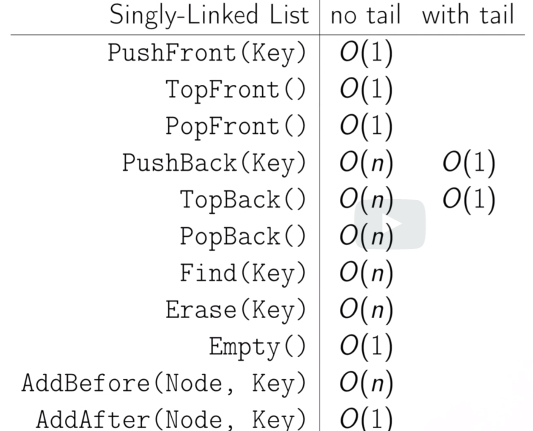
Linked List:

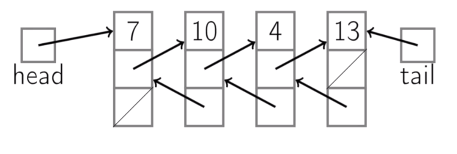


1. Node contains: key and next pointer
2. Operations



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

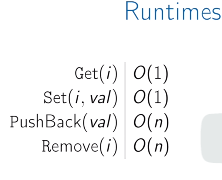
Tree:

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)

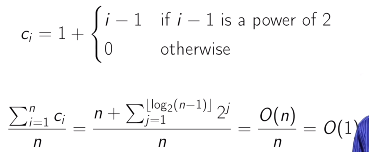
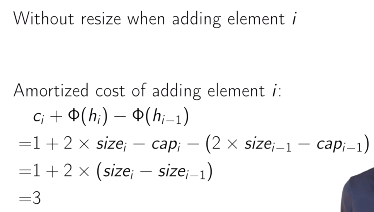
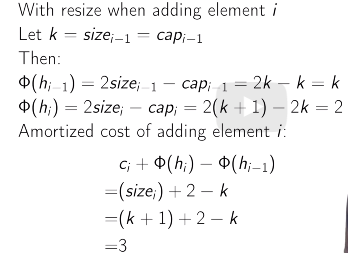
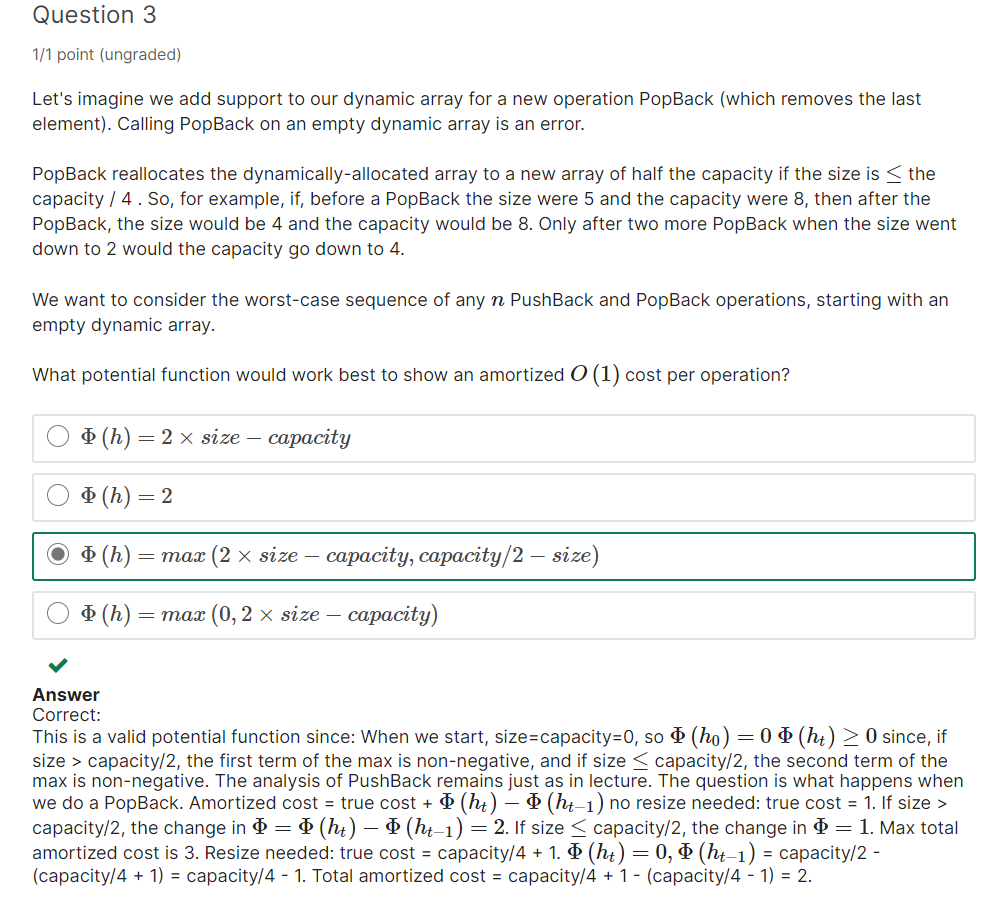
Tree Traversal:

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

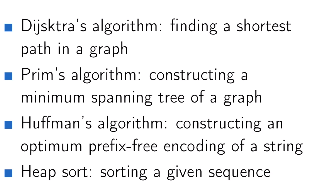
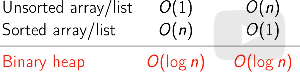
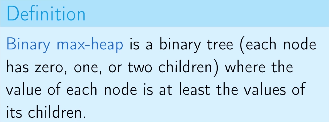
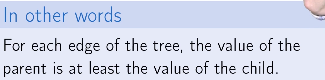
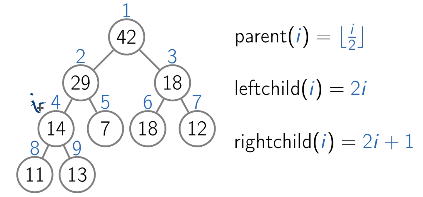
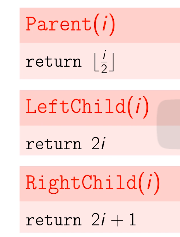
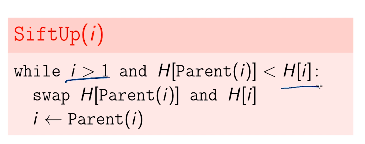
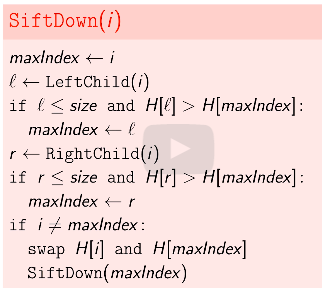
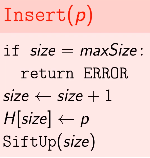
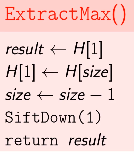
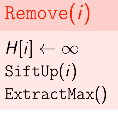
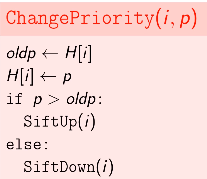
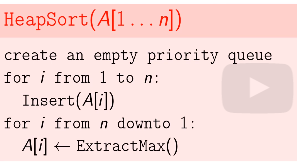
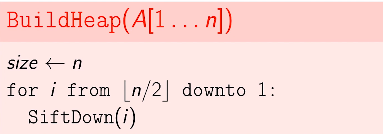
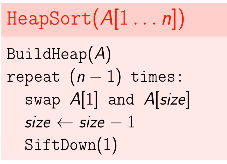
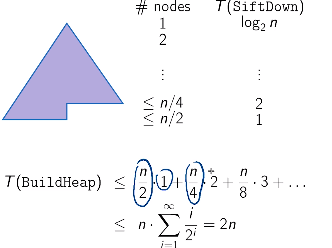
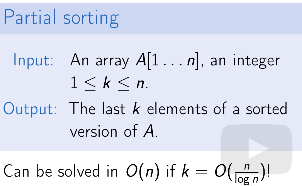
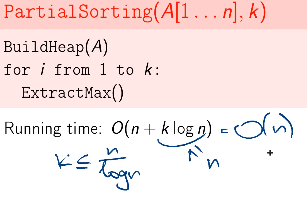
Dynamic Array:

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

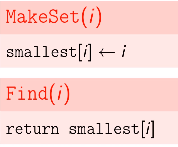
Amortized Analysis:

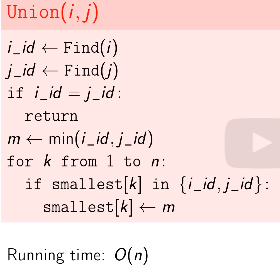
1. Most time O(1) sometime O(n)
2. Amortized cost = Cost(n operations)/n
3. N calls to pushback (aggregate method):
4. 
5. Banker’s method
   1. Charge extra for each cheap operations
   2. Save extra charge as tokens in data structure
   3. Use the tokens to pay for operations
   4. Charge 3 for each 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. 

Priority Queue:

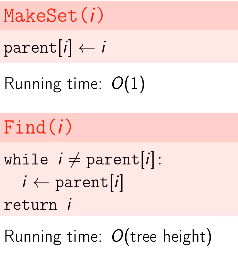
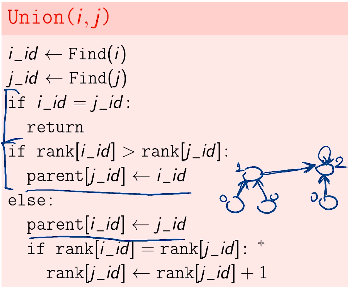
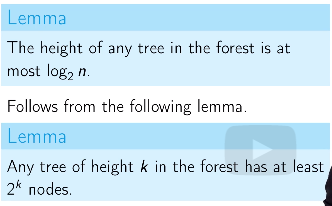
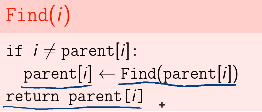
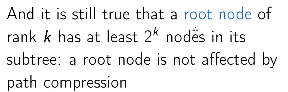
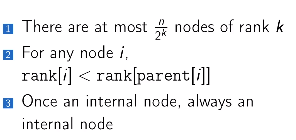
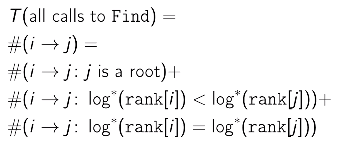
1. Python heapq
2. Element is assigned a priority and come out in order by priority
3. Use case:
   1. Want to process job one by one in order of decreasing priority
   2. New job: Insert(job)
   3. Done job: ExtractMax()
4. Insert(p), ExtractMax(), Remove(it), GetMax(), ChangePriority(it, p)
   1. Elements stored order is not important
5. Algorithm:
   1. 
6. Implementation:
   1. Unsorted Array/List
      1. Insert(e), O(1)
      2. ExtractMax(), O(n)
   2. Sorted Array
      1. ExtractMax(), O(1)
      2. Insert(e): find position O(logn), shift all elements to right O(n), insert O(1). O(n) in total
   3. Sorted double linked List
      1. ExtractMax(), O(1)
      2. Insert(), insert element O(1), find position O(n) (cannot use binary search for list). O(n) in total
      3. 
7. Binary max-Heaps
   1. 
   2. 
   3. Height;
      1. Number of edges on the longest path from the root to a leaf
   4. Operations
      1. GetMax(), return root O(1)
      2. Insert(e):
         1. Attach to a leaf
         2. SiftUp, swapping 2 the node and its parent if the attached is larger, repeat until being binary heap O(tree height)
      3. ExtractMax()
         1. Swap root and a leaf
         2. SiftDown, replace the parent by its larger child if current root is smaller than the larger one. Repeat until satisfied (O(tree height))
      4. ChangePriority
         1. Use SiftUp or SiftDown O(tree height)
      5. Remove:
         1. Change the priority to infinity
         2. Then ExtractMax()
   5. Complete Binary Tree
      1. Keep the height small
      2. Complete: all its levels are filled except possibly the last one which is filled from left to right
      3. Height = O(logn)
      4. Store as Array
         1. 
      5. Cost: keep completeness
      6. Operations change shape
         1. Insert(e)
            1. Insert at the leftmost vacant position in the last level
         2. ExtractMax()
            1. replace root by the last leaf of the last level
   6. Pseudo-code:
      1. maxSize
      2. Parent(), LeftChild(), RightChild()
      3. 
      4. 
      5. 
      6. 
      7. 
      8. 
8. Heap Sort:
   1. 
   2. O(nlogn), generalization of selection sort but smart data structure
   3. Use additional space for priority queue, not in-place
   4. In-place heap sort:
      1. repair heap property from bottom to top O(nlogn)
      2.  This is in-place
      3. Intrasort algorithm
         1. First run quicksort
         2. If slow (exceed clog(n) for some c), switch to heapsort, guaranteed O(nlogn)
9. Building running time
   1. If a node is close to the leaf, sifting down is faster, we have many such nodes
   2. 
   3. 
   4. 
10. Binary Min-Heap
    1. Minimum priority
    2. Generalized to d-ary Heap, height = log\_d(n)

Disjoint Sets

1. Operations
   1. MakeSet(x): create a singleton set {x}
   2. Find(x)
      1. Find(x) = Find(y) if in the same set
   3. Union(x, y), merge two sets containing x and y
2. Naïve implementation
   1. 
   2. Assume the small value of the set as set id



* 1. Use linked list:
     1. Tail as set id
     2. Pro
        1. Union O(1)
        2. Well-defined id
     3. Con
        1. Find O(n) to traverse the list
        2. Union(x, y) O(1) only if we can get the tail of x and head of y in O(1)
     4. Use tree for disjoint set

1. Tree for disjoint set:
   1. Disadvantage of list:
      1. Merge leads to larger list
   2. Trees stored in array
   3. 
   4. Merge:
      1. Link root to the other root, we would like to keep the trees shallow (union by rank heuristic)
   5. Rank[i] is the height of the subtree with root i
   6. 
   7. 
   8. Union and Find by rank heuristic is O(logn)
2. Path compression
   1.  reattach nodes on path to the root
   2. Log\*(n)
3. Path compression and rank heuristic, operation average running time is nearly constant
   1. Height <= Rank
   2. 
   3. 
   4. 
   5. 1st term O(m), 2ndterm O(mlog\*n), 3rd term O(nlog\*n)