CS5800 Algorithms

Module 4. Heap and Binary Search Tree

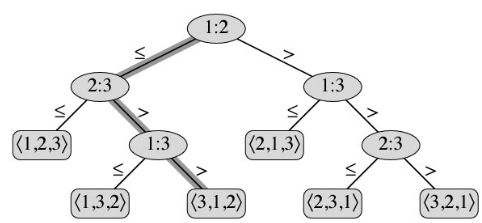
Lower Bounds For Sorts

How Fast Can We Do Sorting By Comparing

Comparison Sorts

- All sorting algorithms we learned so far are based on:
  - Repeatedly *comparing* two elements of the given array.
- We've seen as good as $\Theta(n \lg n)$ comparison sorting algorithms.
  - Merge sort (all cases), quicksort (average/expected cases), heapsort (will be covered later, all cases)
- Is there any better comparison sorting algorithms?
  - Surprisingly (or not) no.
  - It's proven by analyzing any comparison-based sorting algorithm:
    - A sequence of comparisons, determining the final total order.
    - Starting from one pair, its comparison determining next comparison, …
      - We get a so-called decision tree.

Lower Bound For Worst Case



- "Takes at least this long in the worst case"
  - The height of the decision tree!
- There are $n!$ permutations for any given input array of size n.
  - Every permutation must show up as a leaf in the decision tree:
    - $n! \leq l$ (l is the number of leaves in the decision tree)
  - For a binary tree of height h, there are at most 2^h leaves:
    - $l \leq 2^h$
- Therefore, we get $n! \leq 2^h$.
- Solving for h, we get:
  - $h \geq \log_2(n!) = \log_2 n + \log_2(n-1) + \cdots = \Omega(n \lg n)$ (eq. (3.19) in pp. 58)

Sorting In Linear Time

Do We Always Have To Compare To Sort?

5

Counting Sort

- When there are a lot more elements than possible distinct values
  - E.g.: 1,0,2,0,0,1,1,2,0,1,2,0 ← Only 3 possible distinct values, but 12 elements
- Count the number of occurrences of each value, create the "counts" array:
- Then reproduce the sorted sequence out of the counts
- Experiment counting sort at http://visualgo.net/sorting

6

Example: 2, 5, 3, 0, 2, 3, 0, 3 (CLRS Fig. 8.2)

```
COUNTING-SORT(A, B, k)
let C[0 . . k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

	0	1	2	3	4	5
C	2	2	4	6	7	8

Q: What is the counts array content after the first pass of the counting sort algorithm is run on the input array [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2] ?

a) [2, 2, 2, 2, 1, 0, 2]

b) [0, 1, 2, 3, 4, 5, 6]

c) [0, 0, 1, 1, 2, 2, 3, 3, 4, 6]

d) [0, 1, 2, 3, 4, 6]

Counting Sort Time Complexity

- Initializing counts array: $\Theta(k)$ (k is the largest possible value)
- Counting/constructing part: $\Theta(n)$
- Therefore, $\Theta(k + n)$.
- If $k = O(n)$, then $\Theta(n)$.
  - The premise ($k = O(n)$) is important!
  - If k is arbitrarily large (e.g., a double value) and n is not that big (e.g., 100), you don't want to use this algorithm!
- CLRS pp. 195 COUNTING-SORT() pseudocode
  - More involved to meet the "stability" requirement
    - Important for radix sort.

Radix Sort

- Sort *discrete* values digit-by-digit repeatedly in d passes
  - However, start from least-significant digit, and move up! (Counterintuitive)
- Experiment radix sort at http://visualgo.net/sorting
- Example: 329, 457, 657, 839, 436, 720, 355
- Why does it work? How to prove? Use induction on # digits
  - "Stability" in digit-by-digit sorting is important!
- Time complexity: $\Theta(d(n + k))$. If d and k are constants, it's $\Theta(n)$.
  - d: the number of digits
  - k: possible digits (for binary numbers, k=2)
- What about d-bit binary numbers?
  - Θ(d*(n+2)) = Θ(d*n) = Θ(n*log n)

Q: Given array [29, 57, 47, 39, 36, 20, 55], what is the resulting array after the first pass of the radix sort is completed?

a) [20, 55, 36, 57, 47, 29, 39]

b) [20, 29, 36, 39, 47, 55, 57]

c) [20, 55, 36, 47, 57, 29, 39]

d) [29, 20, 39, 36, 47, 57, 55]

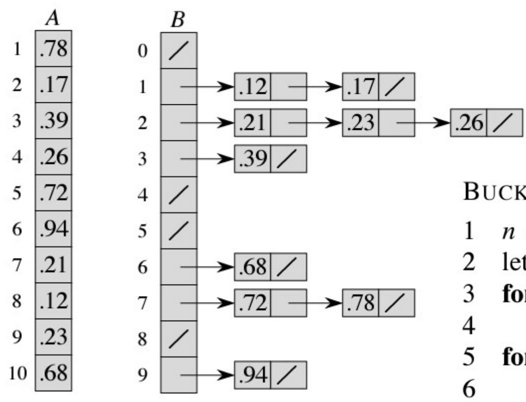
Bucket Sort

- Only good for input array when its values are ***uniformly distributed*** over the interval [min,max]
  - Divide the interval into n equal-sized subintervals, or "buckets"
  - Distribute the n input numbers into the buckets
    - Because of the "uniformly distributed" assumption, each bucket shouldn't contain too many elements
    - Thus, sorting elements in each bucket should be bound to a constant.
  - Final sorting is to collect elements from each bucket one-by-one after sorting elements in each bucket.
- Time complexity analysis: Another probability & random var. analysis
  - $\Theta(n)$, on average, again only under the ***uniformly distributed*** assumption

Bucket Sort Example And Code (CLRS Fig. 8.4)

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \end{aligned}$$



$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n$$

```
BUCKET-SORT(A)
n = A.length
let B[0..n-1] be a new array
for i = 0 to n-1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n-1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n-1] together in order
```


$E\left[n_i^2\right] = 2 - \frac{1}{n}$ (Why?)

- $p = \frac{1}{n}, q = 1 - \frac{1}{n}$
- $\mathrm{Var}[n_i] = n \cdot p \cdot q = n \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right) = 1 - \frac{1}{n}$
- $E[n_i] = n \cdot p = n \cdot \frac{1}{n} = 1$
- $E\left[n_i^2\right] = Var[n_i] + (E[n_i])^2 = 1 - \frac{1}{n} + 1 = 2 - \frac{1}{n}$

Information Retrieval With Elementary Data Structures

Recapping Insertion/Deletion/Search Algorithms With Arrays And Lists

Elementary Information Retrieval

- Sequence of operations of mixed types
  - Insertion/deletion/search of items
- Collection of items: Accessed by an attribute (key)
  - Managed as arrays, linked lists (should be familiar to all already)
  - Binary search trees for better performance
- Time complexities of those operations on different data structures

Elementary Data Structures

- Stacks, queues, linked lists: Undergrad prerequisites
  - Study CLRS Ch. 10 for recap
  - Focus on linked lists for general information retrieval operations (insert/delete/search)
  - Everyone should be able to write code for insert/delete/search on singly/doubly linked lists with *pointers*
- Binary tree representation using *pointers* (CLRS 10.4)
- Time complexities of insert/delete/search algorithms on sorted/unsorted arrays/linked lists
  - Everyone should be able to derive all these

Worst Case Insert/Delete/Search

(A: Array, L: linked list, i: index in array, n: node in list, k: key.)

Operations	Unsorted arrays	Sorted arrays	Unsorted singly linked lists	Sorted singly linked lists	Unsorted doubly linked lists	Sorted doubly linked lists
INSERT(A/L, i/n)	O(n)		O(n)		O(1)	
INSERT(A/L, k)	O(n)	O(n)	O(1)	O(n)	O(1)	O(n)
DELETE(A/L, i/n)	O(n)		O(n)		O(1)	
DELETE(A/L, k)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
SEARCH(A/L, k)	O(n)	O(lg n)	O(n)	O(n)	O(n)	O(n)
MINIMUM(A/L)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)
MAXIMUM(A/L)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)

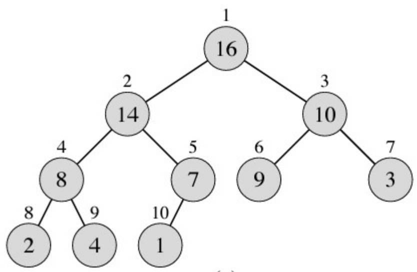
Heaps And Heapsort

When We Want O(1) MAXIMUM() (Or MINIMUM()) All The Time

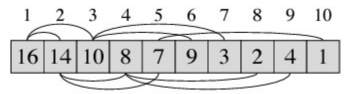
What Is A Heap?

- A data structure that's specialized for retrieving minimum (or maximum) in $O(1)$ time.
  - Many applications for "priority queues" in many other algorithms
  - BST can only give us $O(\lg n)$ (Even balanced BST for worst case)
- Utilize binary tree, but make sure it's as balanced as possible
  - *Complete* binary tree
    - As balanced as possible, all leaves packed to the left
  - With heap property
    - For each node, its value is less than (for min-heap) or great than (for max-heap) both of its children
  - Implemented using an array
    - No need for pointer operations/traversals

Max Heap Example (CLRS Fig. 6.1)



(a)



(b)

$A[\textsc{Parent}(i)] \geq A[i]$

$\textsc{Parent}(i)$
1 **return** $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$
1 **return** $2i$

$\textsc{Right}(i)$
1 **return** $2i + 1$

Q: Which is not a heap?

Figure 1

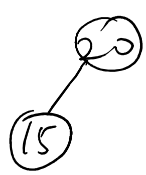


Figure 2

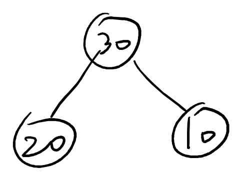


Figure 3

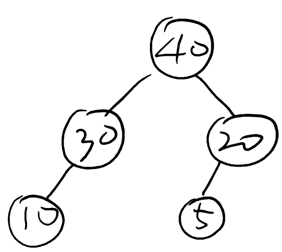


Figure 4

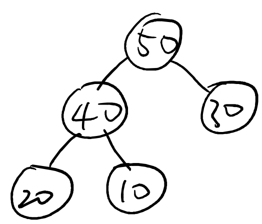


Figure 5

Q: Given the array representation of a max-heap [16, 14, 10, 8, 7], what is the correct array representation of the resulting heap after a new value 19 is inserted?

a) [16, 14, 10, 8, 7, 19]

b) [19, 16, 14, 10, 8, 7]

c) [19, 14, 16, 8, 7, 10]

d) [19, 16, 10, 14, 7, 8]

Q: Given the array representation of a max-heap [19, 14, 16, 8, 7, 10], what is the correct array representation of the resulting heap after its maximum is extracted

a) [14, 16, 8, 7, 10]

b) [16, 14, 10, 8, 7]

c) [19, 14, 8, 7, 10]

d) [14, 8, 16, 7, 10]

Building A Max-Heap

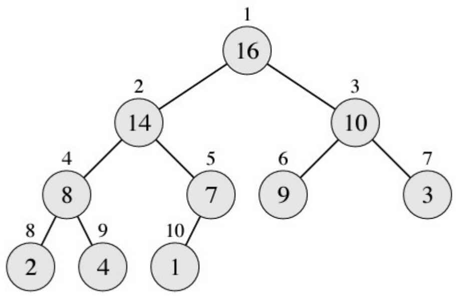
- Given an array of arbitrary values, build a max-heap.
- Two approaches:
  - Insert item by item starting from an empty heap
    - After each insertion, the resulting array must form a max-heap.
    - So fix up each inserted (appended) item by “trickling-up”.
    - n insertions, each insertion possibly taking $O(h)$, resulting in $O(n \lg n)$
  - Consider the original array as a heap
    - Of course it’s not really a heap, so fix one-by-one, from bottom up, but we do “trickling-down” here.
    - Each fix-up could possibly take $O(h)$, and there are n fix-ups possible, so this looks like another $O(n \lg n)$
    - Turns out that this is not a tight bound. It’s actually $O(n)$.
      - Analysis in CLRS 6.3

Building Max-Heap By Item-By-Item Insertions

- Given array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$,

Building Max-Heap By Node-By-Node Fix-ups

- Given array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$,




```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```


Time Complexity Of BUILD-MAX-HEAP(A)

- Naïve/loose analysis: $O(\lg n)$ for each MAX-HEAPIFY(A, i), $n/2$ times, so easily $O(n \lg n)$, but this is not tight as shown below:
- Note that MAX-HEAPIFY(A, i) is not on the root (at height $h = \lfloor \lg n \rfloor$) all the time, but mostly on nodes at lower heights!
  - Up to $n/2$ nodes at height 0 (leaf), $n/4$ nodes at height 1, $n/8$ nodes at height 2, … ➔ Up to $\lceil n/2^{h+1} \rceil$ nodes at height h, where $0 \le h \le \lfloor \lg n \rfloor$
- Therefore, actual # operations is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

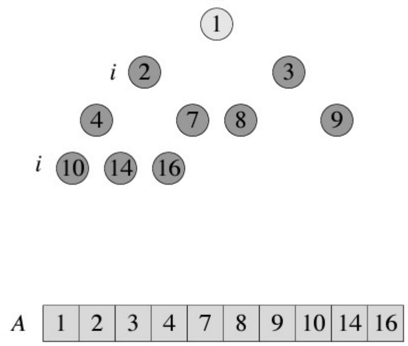
$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O(n) .$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Heapsort By Repeatedly Deleting (Extracting) Max (CLRS Fig. 6.4)

- The root of a max-heap is always the maximum of all values!
  - Remove root. Its sorted position is that of the last node of the heap.
  - Move last node in heap to root, fix-up the heap (trickle-down)
  - Then repeat this whole process until there's no node left in the heap.
- Complexity: $O(n \lg n)$ obviously.
- Experiment all heap operations at http://visualgo.net/heap



Heap As Priority Queue

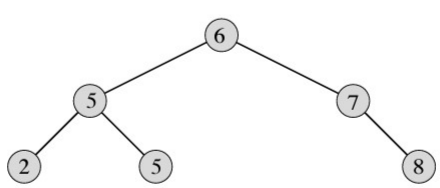
- INSERT(S, x)
  - Insert x into queue so that GET-MAX() and EXTRACT-MAX() is efficient.
  - Place x at the end of array (last node in the heap), trickle it up. $O(\log n)$.
- GET-MAX(S): Always root. $O(1)$.
- EXTRACT-MAX(S): Removes & returns max of all values in queue
  - Remove root, move last heap node to root, trickle it down. $O(\log n)$.
- Many applications in various computer science specialty areas
  - Especially in scheduling & simulation: All about temporal priorities.
  - Also used frequently in many graph algorithms (e.g., shortest paths)

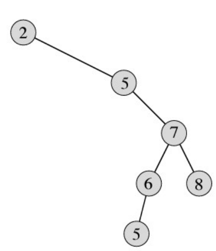
Binary Search Trees

Average-Case Logarithmic Insert/Delete/Search/Minimum/Maximum Operations

What Is A Binary Search Tree (BST)?

- Recursive definition
  - An empty tree is a BST.
  - A binary tree with root node r is a BST if and only if:
    - r's left/right subtree is a BST.
    - All values in r's left subtree are less than or equal to r.
    - All values in r's right subtree are greater than ("or equal to" included in CLRS) r.





Q: Which is not a BST?

Figure 1

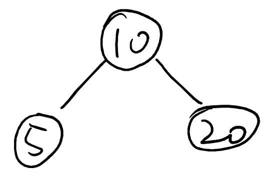


Figure 2

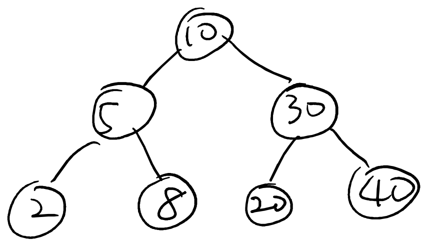


Figure 3

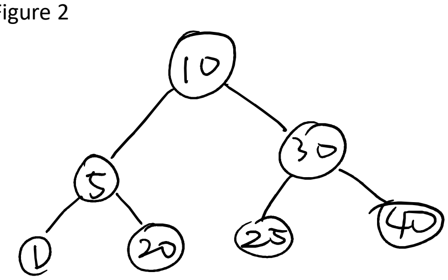
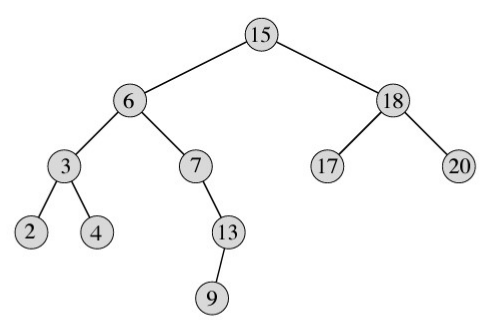


Figure 4

Querying Binary Search Tree

- Searching for a key
  - Very similar to binary search of a sorted array
    - The mid entry is just replaced with the tree node.
    - left = mid + 1: Traversing to the right subtree
    - right = mid – 1: Traversing to the left subtree
- Experiment BST searches at http://visualgo.net/bst
- All are $O(h)$, where h is the tree height.



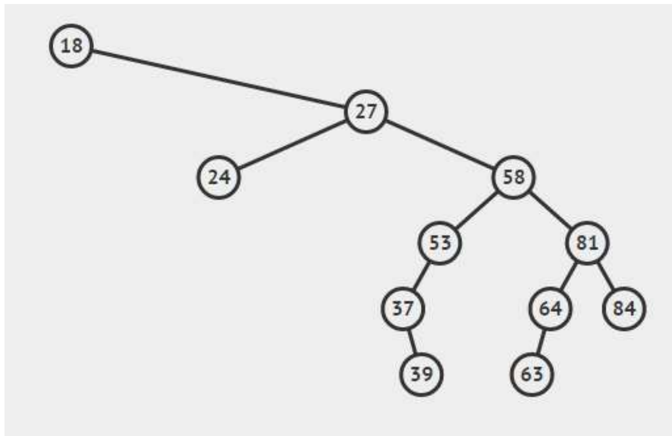
Inserting To Binary Search Tree

- Add a new leaf that continues to meet the BST property
- Start like search, but don't stop at a match
  - Continue until hitting a nil node
  - Add a new leaf there with the inserted value.
- http://visualgo.net/bst
- Still $O(h)$.

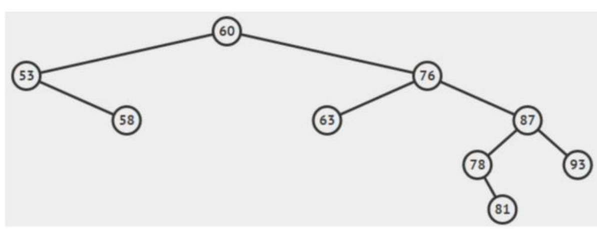
Deleting From Binary Search Tree

- Of course search first. Return if not found.
- If the found node (call it z) is a leaf, trivial.
- If z has only one child, almost trivial.
- If z has both children,
  - Find z's right subtree's minimum (z's successor). Call it y.
  - y should be moved to z's position.
  - Filling in y's vacancy is almost trivial, as y must have no left child.
- Experiment at http://visualgo.net/bst
- Actual code (even pseudocode) can be tricky. Study CLRS 12.3 code.

BST Deletion Examples



Q: Delete 76



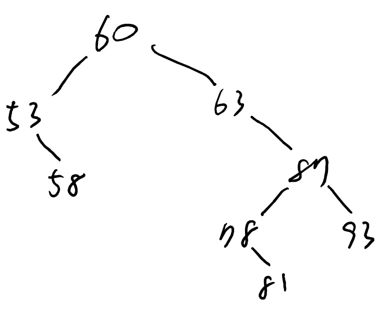


Figure 1

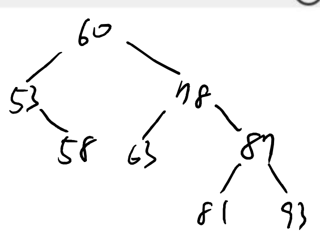


Figure 2

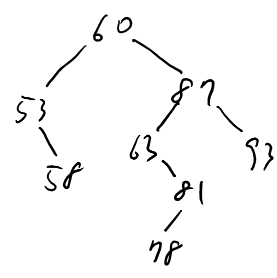


Figure 3

Time Complexities of BST Operations

- All are $O(h)$.
  - $h = n - 1$ in the worst case.
    - Totally skewed to one side, or zig-zag
    - Therefore, worst case BST operations are all $\Theta(n)$.
- Average case tree height
  - Expected height of a randomly built BST
  - Another probability and random variable analysis
    - See Proof of Theorem 12.4 in CLRS pp. 300-303
- Theorem 12.4: Expected height of a randomly built BST on n distinct keys is $O(\lg n)$.

Average Case Insert/Delete/Search

Operations	Unsorted arrays	Sorted arrays	Unsorted singly linked lists	Sorted singly linked lists	Unsorted doubly linked lists	Sorted doubly linked lists	BST (balanced/ average)
INSERT(A/L, i/n)	$O(n)$		$O(n)$		$O(1)$		
INSERT(A/L, k)	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(\lg n)$
DELETE(A/L, i/n)	$O(n)$		$O(n)$		$O(1)$		
DELETE(A/L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$
SEARCH(A/L, k)	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$
MINIMUM(A/L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(\lg n)$
MAXIMUM(A/L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(\lg n)$