Algorithms - CheatSheet

IN BA4 - Ola Nils Anders Svensson

Notes by Ali EL AZDI

This is a cheat sheet for the Algorithms midterm exam. For suggestions, contact me on Telegram (`elazdi_al`) or via EPFL email (`ali.elazdi@epfl.ch`).

March 25th, 2025

Master Theorem
If $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1$, $b > 1$, and $f(n)$ asymptotically positive.
Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
Case 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
Case 3: If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.
Common case - if $f(n) = \Theta(n^d)$ for some exponent d:
- If $\frac{a}{b^d} < 1$ (or $d > \log_b a$), then $T(n) = \Theta(n^d)$.
- If $\frac{a}{b^d} = 1$ (or $d = \log_b a$), then $T(n) = \Theta(n^d \log n)$.
- If $\frac{a}{b^d} > 1$ (or $d < \log_b a$), then $T(n) = \Theta(n^{\log_b a})$.

Queue Operations
Queue-Empty(Q): *Time: O(1), Auxiliary Space: O(1)*
1. Returns TRUE if the queue is empty (Q.head = Q.tail).
2. Returns FALSE otherwise.
Enqueue(Q, x): *Time: O(1), Auxiliary Space: O(1)*
1. Adds element x to the rear of queue Q.
2. Q[Q.tail] = x
3. Q.tail = Q.tail + 1 (or wrap around if using circular array)
Dequeue(Q): *Time: O(1), Auxiliary Space: O(1)*
1. If Queue-Empty(Q), return error "underflow".
2. Otherwise, remove and return the element at the front.
3. x = Q[Q.head]
4. Q.head = Q.head + 1 (or wrap around)
5. Return x
Queue Implementation:
1. Q.head: Index of the front element
2. Q.tail: Index where next element will be inserted
3. In a circular array, indices wrap around
4. Leave one slot empty to distinguish full/empty states
Overall Space Complexity: $O(n)$ for a queue of capacity n

Merge Sort
1. **Divide**: Split the array evenly into two smaller subarrays, and continue dividing recursively.
2. **Sort (Recursively)**: Apply merge sort recursively on each subarray until each has only one element (base case).

```
MERGE-SORT(A, p, r)
if p < r
  q ← ⌊(p + r)/2⌋
  MERGE-SORT(A, p, q)
  MERGE-SORT(A, q + 1, r)
  MERGE(A, p, q, r)
```

3. **Merge**: Combine the two sorted subarrays into a single sorted array:
a) Initializing pointers at the start of each subarray.
b) Comparing the elements pointed to, and appending the smaller one into a new array.
c) Advancing the pointer in the subarray from which the element was chosen.
d) Repeating this process until all elements in both subarrays are merged into the sorted array.

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
let L[1…n1 + 1] and R[1…n2 + 1] be new arrays
for i ← 1 to n1 do
  L[i] ← A[p + i − 1]
for j ← 1 to n2 do
  R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r do
  if L[i] ≤ R[j]
    A[k] ← L[i]
    i ← i + 1
  else
    A[k] ← R[j]
    j ← j + 1
```

Merge Cost Complexity: $O(n)$ per merge operation.
Time Complexity: $O(n \log n)$
Space Complexity: $O(n)$

Priority Queue
Maintains a dynamic set of elements with associated priority values (keys).
Maximum(S): Return element of S with highest priority (return A[1], complexity $O(1)$)
Insert(S,x): Insert element x into set S
1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to key using Heap-Increase-Key
Extract-Max(S): Remove and return element of S with highest priority
1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node
5. Return the copy of the maximum element
Increase-Key(S,x,k): Increase the value of element x's key to the new value k
1. Make sure key ≥ A[i]
2. Update A[i]'s value to key
3. Traverse the tree upward comparing new key to the parent and swapping if necessary
Time Complexity: Insert, Extract-Max, Increase-Key: $O(\log n)$ Maximum: $O(1)$
Space Complexity: $O(n)$

Stack Operations
Stack-Empty(S): *Time: O(1), Auxiliary Space: O(1)*
1. Returns TRUE if the stack is empty.
2. Returns FALSE otherwise.
Push(S, x): *Time: O(1), Auxiliary Space: O(1)*
1. Adds element x to the top of stack S.
2. Increments the stack pointer.
Pop(S): *Time: O(1), Auxiliary Space: O(1)*
1. If Stack-Empty(S), return error "underflow".
2. Otherwise, remove and return the top element.
3. Decrements the stack pointer.
Stack Implementation:
1. Elements are stored in a simple array
2. S.top: Index of the topmost element
3. An empty stack has S.top = 0 or S.top = -1 (implementation dependent)
Overall Space Complexity: $O(n)$ for a stack of size n

Dynamic Programming
Problem: Optimal solutions to overlapping subproblems
Fibonacci Sequence:
Top-Down (Memoization):
1. Create memo array $F[0...n]$ initialized to NIL
2. Base cases: $F[0] = 0$, $F[1] = 1$
3. Recursive with memo:
- Return $F[n]$ if already computed
- Otherwise compute
$F[n] = F[n-1] + F[n-2]$
- Store result in $F[n]$ and return

```
MEMOIZED-FIB(n)
Let r = [0…n] be a new array
for i = 0 to n do
  r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)
if r[n] ≥ 0
  return r[n]
if n = 0 or n = 1
  ans ← 1
else
  ans ← MEMOIZED-FIB-AUX(n − 1, r) + MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```

Bottom-Up (Tabulation):
1. Create array $F[0...n]$
2. Base cases: $F[0] = 0$, $F[1] = 1$
3. For $i = 2$ to n:
- Compute $F[i] = F[i-1] + F[i-2]$
4. Return $F[n]$

```
BOTTOM-UP-FIB(n)
Let r be a new array of size n + 1
r[0] ← 1
r[1] ← 1
for i = 2 to n do
  r[i] ← r[i − 1] + r[i − 2]
return r[n]
```

Time Complexity: $O(n)$
Space Complexity: $O(n)$
(includes input and memo array)

Cut-Rod Problem:
Find optimal way to cut rod to maximize revenue
Top-Down (Memoization):
1. Create memo array $r[0...n]$ with $r[0] = 0$
2. For uncalculated $r[j]$, compute:
$r[j] = \max_{1 \leq i \leq j}(p[i] + r[j-i])$
3. Return $r[n]$

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
  return r[n]
if n == 0
  q = 0
else
  q ← −∞
  for i = 1 to n do
    q ← max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q

MEMOIZED-CUT-ROD(p, n)
let r[0…n] be a new array
for i = 0 to n do
  r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Bottom-Up (Tabulation):
1. Create array $r[0...n]$ with $r[0] = 0$
2. For $j = 1$ to n:
- Compute $r[j] = \max_{1 \leq i \leq j}(p[i] + r[j-i])$
3. Return $r[n]$

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0…n] and s[0…n] be new arrays
r[0] ← 0
for j ← 1 to n do
  q ← −∞
  for i ← 1 to j do
    if q < p[i] + r[j − i]
      q ← p[i] + r[j − i]
      s[j] ← i
  r[j] ← q
return r, s
```

Time Complexity: $O(n^2)$
Space Complexity: $O(n)$
(includes input prices and array)

Akra–Bazzi, Ali Najib Variation:
For recurrence $T(n) = \alpha T(an) + \beta T(bn) + \Theta(n^d)$,
where $\alpha, \beta > 0$, $a, b \in (0,1)$, $d \geq 0$, and unique p with $\alpha a^p + \beta b^p = 1$
Then:
$$p > d \Rightarrow T(n) = \Theta(n^p),$$
$$p = d \Rightarrow T(n) = \Theta(n^d \log n),$$
$$p < d \Rightarrow T(n) = \Theta(n^d).$$
If p is not easily found, compare $\alpha a^d + \beta b^d$ with 1:
$$\alpha a^d + \beta b^d < 1 \Rightarrow T(n) = \Theta(n^d),$$
$$\alpha a^d + \beta b^d = 1 \Rightarrow T(n) = \Theta(n^d \log n),$$
$$\alpha a^d + \beta b^d > 1 \Rightarrow T(n) = \Theta(n^p),$$
with p determined by $\alpha a^p + \beta b^p = 1$.

Insertion Sort.
1 - **Select the key**
Begin with the second element (at index 1) as the *key*.
2 - **Compare and Shift**
Compare the key with elements in the sorted section (to its left).
3 - **Shift Elements**
If an element is greater than the key, shift that element one position to the right.
4 - **Insert the Key**
Once an element less than or equal to the key is found (or you reach the start), insert the key immediately after that element.
5 - **Repeat**
Move forward to the next element, treating it as the new key, and repeat until the array is sorted.
Time Complexity: Worst-case $O(n^2)$, Best-case $O(n)$.
Space Complexity: $O(1)$.

```
INSERTION-SORT(A, n)
for j = 2 to n do
  key = A[j] // Insert A[j] in the sorted seq. A[1..j-1].
  i = j - 1
  while i > 0 and A[i] > key do
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = key
```

Strassen's Matrix Multiplication
Divide: Partition each of A, B, C into four $\frac{n}{2} \times \frac{n}{2}$ submatrices:
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$
Conquer: Compute 7 products (recursively on $\frac{n}{2} \times \frac{n}{2}$ matrices):
$M_1 := (A_{11} + A_{22})(B_{11} + B_{22})$,
$M_2 := (A_{21} + A_{22})B_{11}$,
$M_3 := A_{11}(B_{12} - B_{22})$,
$M_4 := A_{22}(B_{21} - B_{11})$,
$M_5 := (A_{11} + A_{12})B_{22}$,
$M_6 := (A_{21} - A_{11})(B_{11} + B_{12})$,
$M_7 := (A_{12} - A_{22})(B_{21} + B_{22})$.
Combine: Assemble the resulting submatrices to form C:
$C_{11} = M_1 + M_4 - M_5 + M_7$,
$C_{21} = M_2 + M_4$,
$C_{12} = M_3 + M_5$,
$C_{22} = M_1 + M_3 - M_2 + M_6$.
Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.81})$
Space Complexity:$O(n^2)$

Binary Search Trees (BST)
BST-Search
1. Start at root
2. If NULL, return NULL
3. If key = root's key, return root
4. If key < root's key, search left
5. If key > root's key, search right

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
  return x
else if k < x.key
  return TREE-SEARCH(x.left, k)
else
  return TREE-SEARCH(x.right, k)
```

Time: $O(\log n)$ avg, $O(h)$ worst
Space: $O(n)$ for tree, $O(h)$ auxiliary
BST-Minimum
1. Start at root
2. If NULL, return NULL
3. Follow left pointers until no left child
4. Return leftmost node

```
TREE-MINIMUM(x)
while x.left ≠ NIL do
  x ← x.left
return x
```

Time: $O(h)$
Space: $O(n)$ for tree, $O(1)$ auxiliary
BST-Maximum
1. Start at root
2. If NULL, return NULL
3. Follow right pointers until no right child
4. Return rightmost node

```
TREE-MAXIMUM(x)
while x.right ≠ NIL do
  x ← x.right
return x
```

Time: $O(h)$
Space: $O(n)$ for tree, $O(1)$ auxiliary
BST-Successor
1. If right subtree exists:
Return minimum in right subtree
2. Otherwise:
Find first ancestor where node is in left subtree

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
  return TREE-MINIMUM(x.right)
y ← x.p
while y ≠ NIL and x = y.right do
  x ← y
  y ← y.p
return y
```

Time: $O(h)$
Space: $O(n)$ for tree, $O(1)$ auxiliary
BST-Insert
1. Create new node z with key
2. Start at root, track parent y = NIL
3. Move down tree (left if key ¡ node key, right otherwise)
4. Once NULL found, link z as child of y
5. If y is NIL, z becomes root
6. Otherwise, insert z as left or right child based on key comparison

```
TREE-INSERT(T, z)
y ← NIL
x ← T.root
while x ≠ NIL do
  y ← x
  if z.key < x.key
    x ← x.left
  else
    x ← x.right
z.p ← y
if y = NIL
  T.root ← z // tree T was empty
else if z.key < y.key
  y.left ← z
else
  y.right ← z
```

Time: $O(h)$
Space: $O(n)$ for tree, $O(1)$ auxiliary
BST-Postorder
1. Recursively traverse left
2. Recursively traverse right
3. Visit current node
(Children first, then root)

```
POSTORDER-TREE-WALK(x)
if x ≠ NIL
  POSTORDER-TREE-WALK(x.left)
  POSTORDER-TREE-WALK(x.right)
  print key[x]
```

Time: $O(n)$
Space: $O(n)$ for tree, $O(h)$ auxiliary
BST-Inorder
1. Recursively traverse left
2. Visit current node
3. Recursively traverse right
(Visits nodes in sorted order)

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  INORDER-TREE-WALK(x.left)
  print key[x]
  INORDER-TREE-WALK(x.right)
```

Time: $O(n)$
Space: $O(n)$ for tree, $O(h)$ auxiliary
BST-Preorder
1. Visit current node
2. Recursively traverse left
3. Recursively traverse right
(Root first, then children)

```
PREORDER-TREE-WALK(x)
if x ≠ NIL
  print key[x]
  PREORDER-TREE-WALK(x.left)
  PREORDER-TREE-WALK(x.right)
```

Time: $O(n)$
Space: $O(n)$ for tree, $O(h)$ auxiliary
BST-Transplant
Replace subtree at u with v:
1. If u is root, set v as root
2. If u is left child, make v left child of u's parent
3. Else make v right child of u's parent
4. Set v's parent to u's parent

```
TRANSPLANT(T, u, v)
if u.p = NIL
  T.root ← v
else if u = u.p.left
  u.p.left ← v
else
  u.p.right ← v
if v ≠ NIL
  v.p ← u.p
```

Time: $O(1)$
Space: $O(n)$ for tree, $O(1)$ auxiliary
BST-Delete
1. If z has no left: transplant right
2. If z has no right: transplant left
3. With both children:
a. Find successor y
b. Handle y's children
c. Replace z with y

```
TREE-DELETE(T, z)
if z.left == NIL
  TRANSPLANT(T, z, z.right) // z has no left child
else if z.right == NIL
  TRANSPLANT(T, z, z.left) // z has just a left child
else// z has two children
  y ← TREE-MINIMUM(z.right) // y is z's successor
  if y.p ≠ z
    TRANSPLANT(T, y, y.right)
    y.right ← z.right
    y.right.p ← y
  TRANSPLANT(T, z, y)
  y.left ← z.left
  y.left.p ← y
```

Time: $O(h)$
Space: $O(n)$ for tree, $O(1)$ auxiliary

Properties:
- Left subtree: all keys ¡ node's key
- Right subtree: all keys ¿ node's key
- Left and right subtrees are also BSTs
- A **Node** has: **key** (value), **left** & **right** (child pointers), **parent** (optional)
- Tree height **h**: length of longest path from root to leaf

Big-O
If $\exists c > 0$ and $\exists n_0 > 0$, $0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0$, $f(n) = O(g(n))$.
Big-Omega
If $\exists c > 0$ and $\exists n_0 > 0$, $0 \leq c \cdot g(n) \leq f(n)\ \forall n \geq n_0$, $f(n) = \Omega(g(n))$.
Big-Theta
If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$.
Little-o
If $\forall c > 0\ \exists n_0 > 0$, $0 \leq f(n) < c \cdot g(n)\ \forall n \geq n_0$, $f(n) = o(g(n))$.
Relations
$f(n) = o(g(n)) \implies f(n) = O(g(n))$
Comparison of Common Functions (Ascending Order)
$O(1) \ll O((\log n)^c) \ll O(n^c)_{0<c<1} \ll O(n) \ll O(n \log n) \ll O(n^c)_{c>1} \ll O(c^n) \ll O(n!) \ll O(n^n)$.

Heap | Root is A[1] | Left(i) = 2i | Right(i) = 2i + 1 | Parent(i) = $\lfloor i/2 \rfloor$ |
Max-Heapify (heapify subtree rooted at i)
1. Starting at the root
2. Compare A[i], A[Left(i)], A[Right(i)]
3. If necessary, swap A[i] with the largest of the two children
4. **Max-Heapify** the swapped child
5. Continue comparing and swapping down the heap until subtree rooted at i is max-heap
Time Complexity: $O(\log(n))$
Space Complexity: $O(n)$ including heap array, $O(1)$ auxiliary
Max-Heap-Insert (insert new key into heap)
1. Increase heap size: A.heap-size = A.heap-size + 1
2. Set the last element to negative infinity: A[A.heap-size] = $-\infty$
3. Call Max-Heap-Increase-Key to update to the correct value
Time Complexity: $O(\log n)$
Space Complexity: $O(n)$ including heap array, $O(1)$ auxiliary
Max-Heap-Increase-Key (increase key at position i)
1. Ensure new key is larger than current: if key < A[i] then error
2. Set A[i] = key
3. Compare with parent and swap if necessary:
while i ¿ 1 and A[Parent(i)] ¡ A[i]
4. Exchange A[i] with A[Parent(i)]
5. Set i = Parent(i) and continue upward
Time Complexity: $O(\log n)$
Space Complexity: $O(n)$ including heap array, $O(1)$ auxiliary
Build-Max-Heap (build a max-heap from an array)
1. Start from the last non-leaf node at index $\frac{n}{2} - 1$
2. Move upwards to the root (index 0) and:
a. **Max-Heapify** the current node
b. Ensure the subtree rooted here satisfies max-heap property
3. Repeat until the root node is processed
4. After completion, array A represents a valid max heap
Time Complexity: $O(n)$
Space Complexity: $O(n)$ including heap array, $O(1)$ auxiliary
Heap Sort
1. Build a Max Heap:
a. Convert the given array into a max heap
b. Start from the last non-leaf node and heapify upwards
c. Ensure each parent node is greater than its children
2. Extract Maximum Elements:
a. Swap the root (maximum value) with the last element
b. Reduce heap size by one to exclude the last element
c. Heapify the root to maintain max heap property
d. Repeat until heap size becomes 1
3. Final Sorted Array:
a. After extraction, the sorted array in ascending order is obtained
b. Maximum elements are placed at the end
Time Complexity: $O(n \log n)$
Space Complexity: $O(n)$ including array, $O(1)$ auxiliary

```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n ∧ A[l] > A[i]
  largest ← l
else
  largest ← i
if r ≤ n ∧ A[r] > A[largest]
  largest ← r
if largest ≠ i
  exchange A[i] with A[largest]
  MAX-HEAPIFY(A, largest, n)

MAX-HEAP-INSERT(A, key, n)
n ← n + 1
A[n] ← −∞
HEAP-INCREASE-KEY(A, n, key)

HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
  error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[Parent(i)] < A[i] do
  exchange A[i] with A[Parent(i)]
  i ← Parent(i)

BUILD-MAX-HEAP(A, n)
for i ← ⌊n/2⌋ to 1 do
  MAX-HEAPIFY(A, i, n)

HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n to 2 do
  exchange A[1] with A[i]
  MAX-HEAPIFY(A, 1, i − 1)

HEAP-EXTRACT-MAX(A,n)
if n < 1
  error "heap underflow"
max ← A[1]
A[1] ← A[n]
n ← n − 1
MAX-HEAPIFY(A,1,n)
return max
```

Maximum Subarray Problem
Problem: Find contiguous subarray with largest sum
Divide and Conquer Approach:
1 - **Divide.**
Split array at midpoint
mid = $\lfloor (\text{low} + \text{high})/2 \rfloor$.
2 - **Conquer.**
Find maximum subarrays recursively
Left max: in A[low...mid]
Right max: in A[mid + 1...high]
Crossing max: spans the midpoint
3 - **Combine:** Return the largest of the three max(left_max, right_max, crossing_max)
4 - **Finding the Crossing Maximum:**
a. Find maximum suffix in left half (from mid down to low)
b. Find maximum prefix in right half (from mid+1 up to high)
c. Crossing max = max suffix + max prefix
Crossing Subarray: The maximum subarray that crosses the midpoint combines the maximum suffix of the left half with the maximum prefix of the right half.
Time Complexity: $\Theta(n \log n)$ due to $T(n) = 2T(n/2) + \Theta(n)$
Space Complexity: $O(\log n)$ for recursion stack

```
FIND-MAX-SUBARRAY(A, low, high)
if high == low
  return (low, high, A[low])
else
  mid = ⌊(low + high)/2⌋
  (left-low, left-high, left-sum) = FIND-MAX-SUBARRAY(A, low, m
  (right-low, right-high, right-sum) = FIND-MAX-SUBARRAY(A, mid
  (cross-low, cross-high, cross-sum) = FIND-MAX-CROSSING-SUBA
low, mid, high)
  if left-sum ≥ right-sum and left-sum ≥ cross-sum
    return (left-low, left-high, left-sum)
  else if right-sum ≥ left-sum and right-sum ≥ cross-sum
    return (right-low, right-high, right-sum)
  else
    return (cross-low, cross-high, cross-sum)

FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum ← −∞,  sum ← 0
for i = mid to low do
  sum ← sum + A[i]
  if sum > left-sum
    left-sum ← sum,  max-left ← i
right-sum ← −∞,  sum ← 0
for j = mid + 1 to high do
  sum ← sum + A[j]
  if sum > right-sum
    right-sum ← sum,  max-right ← j
return (max-left, max-right, left-sum + right-sum)
```

Data Structure Operations Summary
Priority Queue Operations

Operation	Description	Time	Space
Construction	Create priority queue from array of n elements	$O(n)$	$O(n)$
Maximum(S)	Returns element with highest priority	$O(1)$	$O(1)$
Extract-Max(S)	Removes and returns element with highest priority	$O(\log n)$	$O(1)$
Insert(S, x)	Inserts element x into set S	$O(\log n)$	$O(1)$
Increase-Key(S, x, k)	Increases priority of element x to k	$O(\log n)$	$O(1)$

Stack Operations

Operation	Description	Time	Space
Construction	Create stack from array of n elements	$O(n)$	$O(n)$
Stack-Empty(S)	Returns TRUE if stack is empty, FALSE otherwise	$O(1)$	$O(1)$
Push(S, x)	Adds element x to top of stack S	$O(1)$	$O(1)$
Pop(S)	Removes and returns top element from stack S	$O(1)$	$O(1)$

Heap Operations

Operation	Description	Time	Space
Construction	Create heap from array of n elements	$O(n)$	$O(n)$
Max-Heapify(A, i)	Maintains max-heap property at node i	$O(\log n)$	$O(1)$
Build-Max-Heap(A,n)	Converts array A of n elements into max heap	$O(n)$	$O(1)$
Heap-Sort(A, n)	Sorts array A of n elements using heap structure	$O(n \log n)$	$O(1)$
Max-Heap-Insert(A, k)	Inserts key k into heap A	$O(\log n)$	$O(1)$
Heap-Extract-Max(A)	Returns and removes largest element from heap A	$O(\log n)$	$O(1)$
Heap-Increase-Key(A,i,k)	Increases key at index i to new value k	$O(\log n)$	$O(1)$

BST Operations

Operation	Description	Time	Space
Construction	Create BST from array of n elements	Best: $O(n \log n)$, Worst: $O(n^2)$	$O(n)$
BST-Search(T, k)	Finds node with key k in tree T	$O(h)$	$O(h)$
BST-Minimum(T)	Returns node with smallest key in T	$O(h)$	$O(1)$
BST-Maximum(T)	Returns node with largest key in T	$O(h)$	$O(1)$
BST-Successor(x)	Returns node with smallest key greater than x's key	$O(h)$	$O(1)$
BST-Insert(T, z)	Inserts node z into BST T	$O(h)$	$O(1)$
BST-Delete(T, z)	Removes node z from BST T	$O(h)$	$O(1)$
BST-Inorder(T)	Visits all nodes in sorted order	$O(n)$	$O(h)$
BST-Preorder(T)	Visits root before its children	$O(n)$	$O(h)$
BST-Postorder(T)	Visits children before root	$O(n)$	$O(h)$

Queue Operations

Operation	Description	Time	Space
Construction	Create queue from array of n elements	$O(n)$	$O(n)$
Queue-Empty(Q)	Returns TRUE if queue is empty, FALSE otherwise	$O(1)$	$O(1)$
Enqueue(Q, x)	Adds element x to rear of queue Q	$O(1)$	$O(1)$
Dequeue(Q)	Removes and returns front element from queue Q	$O(1)$	$O(1)$

Linked List Operations

Operation	Description	Time	Space
Construction	Create linked list from array of n elements	$O(n)$	$O(n)$
List-Search(L, k)	Returns pointer to first node with key k, NULL if not found	$O(n)$	$O(1)$
List-Insert(L, x)	Inserts node x at beginning of list L	$O(1)$	$O(1)$
List-Delete(L, x)	Removes node x from list L	$O(n)$ find, $O(1)$ del	$O(1)$

Matrix Chain Multiplication:
Find optimal parenthesization to minimize multiplications
Bottom-Up (Tabulation):
1. Create table $m[1...n, 1...n]$ with $m[i,i] = 0$
- $m[i,j]$ stores minimal cost of multiplying matrices i through j
2. For $l = 2$ to n (chain length):
3. For $i = 1$ to $n - l + 1$:
- Set $j = i + l - 1$
- Compute $m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$
- Store k in $s[i,j]$ that achieved minimum cost
4. Return $m[1,n]$

```
MATRIX-CHAIN-ORDER(p)
n ← p.length − 1
let m[1..n, 1..n] and s[1..n, 1..n] be new tables
for i = 1 to n do
  m[i, i] ← 0
for ℓ = 2 to n do // ℓ is the chain length
  for i = 1 to n − ℓ + 1 do
    j ← i + ℓ − 1
    m[i, j] ← ∞
    for k = i to j − 1 do
      q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
      if q < m[i, j]
        m[i, j] ← q
        s[i, j] ← k
return m, s
```

Complexity:
- Time: $O(n^3)$
- Space: $O(n^2)$ (includes matrix dimensions and tables)
- Table $s[i,j]$ stores index of last matrix in
first parenthesized group