Mixed Notes

Math, Finance, CS

Mixed Notes

Math, Finance, CS

Cover: Picture by Anna Sulivan - @aesullivan2010 (Modified)
Style: EPFL Report Style, with modifications by Batuhan Faik Derinbay

Contents

1

Computer Science

1.1. Data Structures

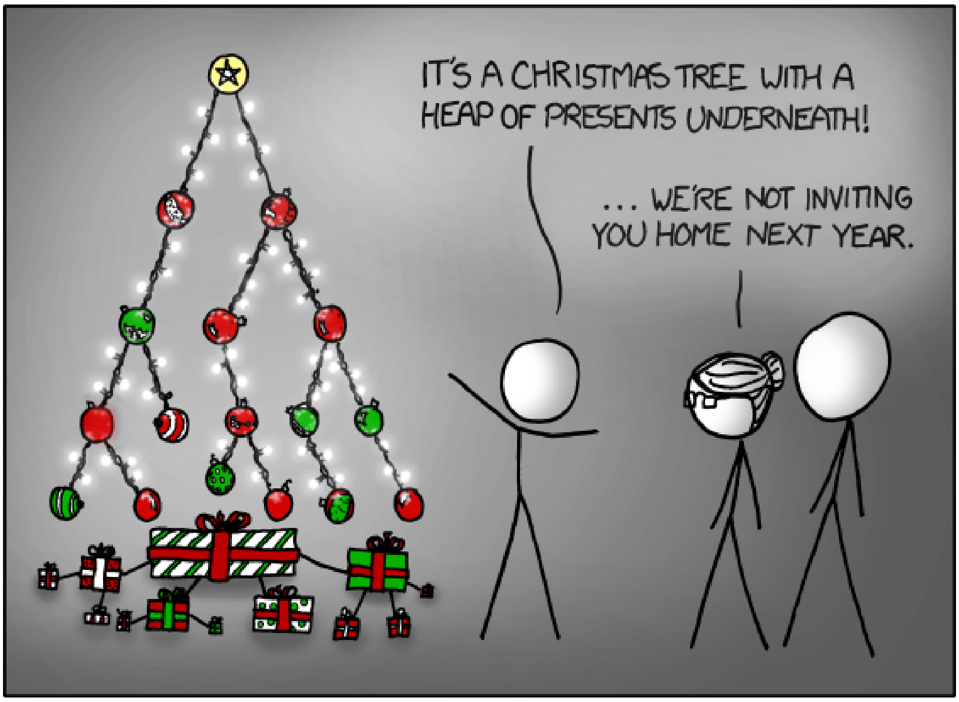

Figure 1.1: *XKCD 835: Tree*

1.1.1. Stack

A stack is a linear last in first out (LIFO) data structure. Elements pushed last are popped first. The major operations supported by a stack are represented below.

Operation	Time Complexity	Description
Push	$O(1)$	Add an element to the top of the stack.
Pop	$O(1)$	Remove the top element from the stack.
IsEmpty	$O(1)$	Check if the stack is empty.
IsFull	$O(1)$	Check if the stack is full.
Peek	$O(1)$	Get the top element without removing it.

Table 1.1: Stack Operations, Their Time Complexities, and Descriptions

```python
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = []

    def push(self, item):
        if not self.is_full():
            self.stack.append(item)
        else:
            print("Stack is full!")

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            print("Stack is empty!")
            return None

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            print("Stack is empty!")
            return None

    def is_empty(self):
        return len(self.stack) == 0

    def is_full(self):
        return len(self.stack) == self.capacity

# Example usage:
stack = Stack(2)
stack.push(1)
stack.push(2)
stack.push(3) # Should print "Stack is full!"
print(stack.pop())  # Should print 3
print(stack.peek())  # Should print 2
print(stack.is_empty())  # Should print False
```

Listing 1.1: Python Stack Implementation

1.1.2. Queue

A queue is a linear first in first out (FIFO) data structure. Elements pushed first are popped first. The major operations supported by a stack are represented below.

Operation	Time Complexity	Description
Enqueue	$O(1)$	Add an element to the end of the queue.
Dequeue	$O(1)$	Remove the top element from the queue.
IsEmpty	$O(1)$	Check if the queue is empty.
IsFull	$O(1)$	Check if the queue is full.
Peek	$O(1)$	Get the top element without removing it.

Table 1.2: Queue Operations, Their Time Complexities, and Descriptions

```python
class Queue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.head = self.tail = -1

    def enqueue(self, data):
        if self.tail == self.capacity - 1:
            print("The queue is full")
        else:
            if self.head == -1:
                self.head = 0
            self.tail += 1
            self.queue[self.tail] = data

    def dequeue(self):
        if self.head == -1:
            print("The queue is empty")
            return None
        temp = self.queue[self.head]
        if self.head == self.tail:
            self.head = self.tail = -1
        else:
            self.head += 1
        return temp

obj = Queue(5)
obj.enqueue(1)
obj.enqueue(2)
obj.dequeue()
```

Listing 1.2: Python Stack Implementation

1.1.3. Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority value. Elements are served based on their priority, with higher priority elements being served first. If elements have the same priority, they are served according to their order in the queue. **Note that priority queues are abstract data types**

Operation	Time Complexity	Description
Insert	$O(\log n)$	Add an element with a priority to the queue.
Delete	$O(\log n)$	Remove the element with the highest priority.
Peek	$O(1)$	Get the element with the highest priority without removing it.

Table 1.3: Priority Queue Operations, Their Time Complexities, and Descriptions

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def insert(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def delete(self):
        return heapq.heappop(self._queue)[-1]

    def peek(self):
        return self._queue[0][-1] if self._queue else None

# Example usage
pq = PriorityQueue()
pq.insert('task1', 1)
pq.insert('task2', 5)
pq.insert('task3', 3)
print(pq.delete())
```

Listing 1.3: Python Stack Implementation

Problem Statement: Design a class to find the Kth largest element in a stream. Note that it is the Kth largest element in the sorted order, not the Kth distinct element.

Approach: Use a min-heap (priority queue) to maintain the K largest elements seen so far.

Algorithm:

1. Initialize a min-heap with a capacity of K.
2. For each new element in the stream, add it to the heap.
3. If the heap size exceeds K, remove the smallest element.
4. The root of the heap will be the Kth largest element.

1.1.4. Deque

A deque (double-ended queue) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

Operation	Time Complexity	Description
Insert Front	$O(1)$	Add an element to the front of the deque.
Insert Rear	$O(1)$	Add an element to the rear of the deque.
Delete Front	$O(1)$	Remove an element from the front of the deque.
Delete Rear	$O(1)$	Remove an element from the rear of the deque.

Table 1.4: Deque Operations, Their Time Complexities, and Descriptions

```python
from collections import deque

class Deque:
    def __init__(self):
        self.deque = deque()

    def insert_front(self, item):
        self.deque.appendleft(item)

    def insert_rear(self, item):
        self.deque.append(item)

    def delete_front(self):
        if self.deque:
            return self.deque.popleft()
        else:
            print("Deque is empty")
            return None

    def delete_rear(self):
        if self.deque:
            return self.deque.pop()
        else:
            print("Deque is empty")
            return None

dq = Deque()
dq.insert_rear(1)
dq.insert_front(2)
print(dq.delete_front())
print(dq.delete_rear())
```

Listing 1.4: Python Stack Implementation

1.1.5. Heap

A heap is a special tree-based data structure that satisfies the heap property. In a max heap, for any given node, the value of the node is greater than or equal to the values of its children. In a min heap, the value of the node is less than or equal to the values of its children.

Operation	Time Complexity	Description
Insert	$O(\log n)$	Add an element to the heap.
Delete	$O(\log n)$	Remove the root element from the heap.
Peek	$O(1)$	Get the root element without removing it.

Table 1.5: Heap Operations, Their Time Complexities, and Descriptions

```python
import heapq

class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, item):
        heapq.heappush(self.heap, item)

    def delete(self):
        return heapq.heappop(self.heap)

    def peek(self):
        return self.heap[0] if self.heap else None
```

Listing 1.5: Python Stack Implementation

1.1.6. Fibonacci Heap

A Fibonacci heap is a collection of trees which follow the min-heap or max-heap property. It has more efficient heap operations than binary heaps and mainly used for Dijkstra's algorithm.

Operation	Time Complexity	Description
Insert	$O(1)$	Add an element to the heap.
Extract Min	$O(\log n)$	Remove the minimum element from the heap.
Decrease Key	$O(1)$	Decrease the value of a key.

Table 1.6: Fibonacci Heap Operations, Their Time Complexities, and Descriptions

Note: The implementation of a Fibonacci heap is quite complex and typically requires a specialized library or extensive custom code. The original paper is referenced here: *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms*

1.1.7. Linked List

A linked list is a linear data structure where each element is a separate object, known as a node. Each node contains two items: the data and a reference (or link) to the next node in the sequence.

Operation	Time Complexity	Description
Insert	$O(1)$	Add an element to the beginning of the list.
Delete	$O(1)$	Remove an element from the beginning of the list.
Search	$O(n)$	Find an element in the list.

Table 1.7: Linked List Operations, Their Time Complexities, and Descriptions

```python
class Node:
    def __init__(self, data):
        self.data, self.next = data, None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        new_node.next, self.head = self.head, new_node

    def delete(self, key):
        temp, prev = self.head, None
        while temp and temp.data != key:
            prev, temp = temp, temp.next
        if temp:
            if prev: prev.next = temp.next
            else: self.head = temp.next

    def search(self, key):
        current = self.head
        while current:
            if current.data == key: return True
            current = current.next
        return False
```

Listing 1.6: Python Linked List Implementation

1.1.8. Doubly Linked List

A doubly linked list is a type of linked list in which each node contains a reference to both the next and the previous node in the sequence.

Operation	Time Complexity	Description
Insert	$O(1)$	Add an element to the beginning of the list.
Delete	$O(1)$	Remove an element from the beginning of the list.
Search	$O(n)$	Find an element in the list.

Table 1.8: Doubly Linked List Operations, Their Time Complexities, and Descriptions

```python
class Node:
    def __init__(self, data):
        self.data, self.next, self.prev = data, None, None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        new_node.next, self.head = self.head, new_node
        if self.head: self.head.prev = new_node
        self.head = new_node

    def delete(self, key):
        temp = self.head
        while temp:
            if temp.data == key:
                if temp.prev: temp.prev.next = temp.next
                if temp.next: temp.next.prev = temp.prev
                if temp == self.head: self.head = temp.next
                return
            temp = temp.next

    def search(self, key):
        current = self.head
        while current:
            if current.data == key: return True
            current = current.next
        return False
```

Listing 1.7: Python Doubly Linked List Implementation

1.1.9. Circular Linked List

A circular linked list is a variation of a linked list in which the last node points back to the first node, forming a circle.

Operation	Time Complexity	Description
Insert	$O(1)$	Add an element to the beginning of the list.
Delete	$O(1)$	Remove an element from the beginning of the list.
Search	$O(n)$	Find an element in the list.

Table 1.9: Circular Linked List Operations, Their Time Complexities, and Descriptions

```python
class Node:
    def __init__(self, data):
        self.data, self.next = data, None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            new_node.next = self.head
        else:
            current = self.head
            while current.next != self.head: current = current.next
            current.next, new_node.next = new_node, self.head

    def delete(self, key):
        if not self.head: return
        current, prev = self.head, None
        while True:
            if current.data == key:
                if prev: prev.next = current.next
                else:
                    if current.next == self.head: self.head = None
                    else:
                        last = self.head
                        while last.next != self.head: last = last.next
                        last.next = current.next
                        self.head = current.next
                return
            prev, current = current, current.next
            if current == self.head: break

    def search(self, key):
        if not self.head: return False
        current = self.head
        while True:
            if current.data == key: return True
            current = current.next
            if current == self.head: break
        return False
```

Listing 1.8: Python Circular Linked List Implementation

1.1.10. Hash Map

A hash map (or hash table) is a data structure that implements an associative array abstract data type, mapping keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Chaining is a common method for handling collisions, where each bucket contains a list of entries that hash to the same index. This allows multiple key-value pairs to be stored in the same bucket.

Operation	Time Complexity	Description
Insert	$O(1)$	Add a key-value pair to the table.
Delete	$O(1)$	Remove a key-value pair from the table.
Search	$O(1)$	Find a value by its key.

Table 1.10: Hash Map Operations, Their Time Complexities, and Descriptions

Note: The worst-case time complexity for hash maps can degrade to $O(n)$ if many collisions occur, leading to long chains in the buckets.

```python
class HashMap:
    def __init__(self):
        self.table = [[] for _ in range(10)]

    def _hash_function(self, key):
        return key % len(self.table)

    def insert(self, key, value):
        index = self._hash_function(key)
        for i, (k, _) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))

    def delete(self, key):
        index = self._hash_function(key)
        self.table[index] = [(k, v) for k, v in self.table[index] if k != key]

    def search(self, key):
        index = self._hash_function(key)
        for k, v in self.table[index]:
            if k == key: return v
        return None
```

Listing 1.9: Python Hash Map Implementation

1.1.11. Tree

A tree is a hierarchical data structure consisting of nodes, where each node has a value and references to child nodes. The top node is called the root, and nodes with no children are called leaves. Trees can degenerate into linked lists in the worst case.

Operation	Time Complexity	Description
Insert	$O(h)$	Add a node to the tree (h is the height).
Delete	$O(h)$	Remove a node from the tree.
Search	$O(h)$	Find a node in the tree.

Table 1.11: Tree Operations, Their Time Complexities, and Descriptions

Note: In the worst case, a tree can degenerate into a linked list, leading to $O(n)$ time complexity for operations.

```python
class TreeNode:
    def __init__(self, value):
        self.value, self.left, self.right = value, None, None

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_rec(self.root, value)

    def _insert_rec(self, node, value):
        if value < node.value:
            node.left = node.left or TreeNode(value)
            if node.left: self._insert_rec(node.left, value)
        else:
            node.right = node.right or TreeNode(value)
            if node.right: self._insert_rec(node.right, value)

    def search(self, value):
        return self._search_rec(self.root, value)

    def _search_rec(self, node, value):
        if not node or node.value == value:
            return node
        return self._search_rec(node.left if value < node.value else node.right, value)
```

Listing 1.10: Python Tree Implementation

1.1.12. Trie

A trie (or prefix tree) is a special type of tree used to store associative data structures. A common application of a trie is storing a predictive text or autocomplete dictionary. Each node represents a character of a string, and paths down the tree represent words.

Operation	Time Complexity	Description
Insert	$O(m)$	Add a word of length m to the trie.
Search	$O(m)$	Find a word of length m in the trie.
Delete	$O(m)$	Remove a word of length m from the trie.

Table 1.12: Trie Operations, Their Time Complexities, and Descriptions

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            node.children.setdefault(char, TrieNode())
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children: return False
            node = node.children[char]
        return node.is_end_of_word

    def delete(self, word):
        self._delete_rec(self.root, word, 0)

    def _delete_rec(self, node, word, index):
        if index == len(word):
            if not node.is_end_of_word: return False
            node.is_end_of_word = False
            return len(node.children) == 0
        char = word[index]
        if char not in node.children: return False
        should_delete = self._delete_rec(node.children[char], word, index + 1)
        if should_delete:
            del node.children[char]
            return len(node.children) == 0
        return False
```

Listing 1.11: Python Trie Implementation

1.1.13. Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree data structure in which each node has at most two children, referred to as the left child and the right child. For each node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater. This property allows for efficient searching, insertion, and deletion operations.

Operation	Time Complexity	Description
Insert	$O(h)$	Add a node to the tree (h is the height).
Delete	$O(h)$	Remove a node from the tree.
Search	$O(h)$	Find a node in the tree.

Table 1.13: Binary Search Tree Operations, Their Time Complexities, and Descriptions

```python
class TreeNode:
    def __init__(self, value):
        self.value, self.left, self.right = value, None, None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_rec(self.root, value)

    def _insert_rec(self, node, value):
        if value < node.value:
            node.left = node.left or TreeNode(value)
            if node.left: self._insert_rec(node.left, value)
        else:
            node.right = node.right or TreeNode(value)
            if node.right: self._insert_rec(node.right, value)

    def search(self, value):
        return self._search_rec(self.root, value)

    def _search_rec(self, node, value):
        if not node or node.value == value:
            return node
        return self._search_rec(node.left if value < node.value else node.right, value)

# Example usage
bst = BST()
bst.insert(5)
bst.insert(3)
bst.insert(7)
print(bst.search(3).value)  # Should print 3
```

Listing 1.12: Python Binary Search Tree Implementation

1.1.14. AVL Tree

An AVL tree is a self-balancing binary search tree where the difference in heights between the left and right subtrees (the balance factor) is at most one for all nodes. This property ensures that the tree remains balanced, providing efficient operations.

Operation	Time Complexity	Description
Insert	$O(\log n)$	Add a node and rebalance the tree.
Delete	$O(\log n)$	Remove a node and rebalance the tree.
Search	$O(\log n)$	Find a node in the tree.

Table 1.14: AVL Tree Operations, Their Time Complexities, and Descriptions

AVL Tree Pseudocode

AVL Tree Operations:
Insert(value):
1. Insert the value as in a regular BST.
2. Update the height of the node.
3. Check the balance factor:
- If balance factor > 1:
- If left child is right-heavy, perform left rotation.
- If left child is left-heavy, perform right rotation.
- If balance factor < -1:
- If right child is left-heavy, perform right rotation.
- If right child is right-heavy, perform left rotation.

Delete(value): 1. Delete the value as in a regular BST. 2. Update the height of the node. 3. Check the balance factor and perform rotations as necessary.

1.1.15. Red-Black Tree

A Red-Black Tree is a type of self-balancing binary search tree where each node has an extra bit for denoting the color of the node, either red or black. This coloring helps maintain balance during insertions and deletions.

Operation	Time Complexity	Description
Insert	$O(\log n)$	Add a node and rebalance the tree.
Delete	$O(\log n)$	Remove a node and rebalance the tree.
Search	$O(\log n)$	Find a node in the tree.

Table 1.15: Red-Black Tree Operations, Their Time Complexities, and Descriptions

Red-Black Tree Pseudocode

Red-Black Tree Operations:
Insert(value):
1. Insert the value as in a regular BST.
2. Color the new node red.
3. While the parent is red:
- If the parent is a left child:
- Check the uncle's color:
- If red, recolor and move up.
- If black, perform rotations.
- If the parent is a right child, do the symmetric case.
4. Ensure the root is black.

Delete(value):
1. Delete the value as in a regular BST.
2. If the node is red, simply remove it.
3. If the node is black, fix the tree to maintain properties.