Basic Data Structures

May 31, 2018
Basic Data Structures

- Array
- Dynamic Array (amortized analysis)
- LinkedList
- Stack
- Queue
- Set
- Map
Array

- Many advantages over linked list
  - Constant-time access for any index
  - Space efficiency: all space is used for data

- Restriction:
  - Inserting a new element in an array of elements is expensive
  - Once allocated, an array has a fixed length

- Solution: dynamic array
Dynamic Array

- Initialize an array with one element
- Before inserting a new element (at the end), if the array is full
  - Allocate a new array of twice the length
  - Copy the existing elements to the new array
- Then proceed with insertion
Amortized analysis

- What is the time complexity of insertion for a dynamic array?
- Amortized analysis is a strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.
- It gives us a worst-case bound on the cost of an algorithm.
- Aggregate method
- Accounting method
Aggregate method

- The cost of the $i$-th insertion is
  \[ c_i = \begin{cases} 
  i & \text{if } i - 1 \text{ is a power of 2} \\
  1 & \text{otherwise}
  \end{cases} \]

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| $c_i$ | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 | 1 | 1 | 1 | 1 | 1 | 17 |

- The total cost of $n$ insertions is
  \[
  \sum_{i=1}^{n} 1 + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n
  \]

- The average cost of one insertion is 3
We will say that the amortized cost for the $i$th insertion is 3 dollars, and this works as follows:

- One dollar pays for inserting the element itself.
- One dollar is stored to move the element later when the array is doubled.
- One dollar is stored to move an element in the array that was already moved from previous array.

For instance, the size of the array is $m$ immediately after expansion. So the number of elements in the array is $m/2$. If we charge 3 dollars for each insertion, then by the time the array is filled up again, we will have $2(m/2)$ extra dollars, which pays for moving all the elements to the new array.
Let $n$ be the length of the array

- Access element of index $i$, $O(1)$
- Insert at the end
  - Amortized $O(1)$ by using dynamic array
- Insert anywhere (to maintain the array as sorted)
  - Best case $O(1)$
  - Worst case $O(n)$
  - Average case $O(n)$
- Delete at the end
  - $O(1)$
- Delete anywhere
  - Best case $O(1)$
  - Worst case $O(n)$
  - Average case $O(n)$
A linked list is an ordered sequence of elements: $A_0, A_1, A_2, \ldots, A_{n-1}$

- Simplest form: singly linked, with a pointer to the head of the list, not sorted
  - Rarely maintained as sorted

- Variations: doubly linked, two pointers (head and tail), circular

- If the size of an element is large, a linked list may be a better choice than an array
/* Java version */
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
    }
}

/* Python version */
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None
Basic Operations

- **Insertion**
  - Inserting B between A and C: B.next = C A.next = B

- **Deletion**
  - Deleting B: A.next = B.next

- **Find**
  ```java
  while (head != null) {
    if (head.val == val) return head;
    head = head.next;
  }
  ```

- **Reverse**
  ```java
  while (currNode != null) {
    nextNode = currNode.next
    currNode.next = prevNode
    prevNode = currNode
    currNode = nextNode
  }
  ```
Time Complexities of Linked List Operations

- Insert (at the front): $O(1)$
- Find
  - Best case $O(1)$
  - Worst case $O(n)$
  - Average case $O(n)$
- Delete
  - Best case $O(1)$
  - Worst case $O(n)$
  - Average case $O(n)$
// two pointers
def removeNthFromEnd(head, n):
    fast = slow = head
    for _ in range(n):
        fast = fast.next
    if not fast: return head.next
    while fast.next:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return head
Stacks support two operations

- Push
- Pop
- Retrieval from stacks is last-in, first-out (LIFO)

Stacks can be easily implemented by either arrays or linked lists

Applications: reversing a word, ”undo” mechanism in text editors, matching braces, etc.
Example: Valid Parentheses

- Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
- Valid: '{[()]}'
- Invalid: '[[(''}
def isValid(s):
    stack = []
    for x in s:
        if x == '(' or x == '{' or x == '[':
            stack.append(x)
        else:
            top = stack.pop()
            if not (top == '(' and x == ')'
                    or top == '[' and x == ']
                    or top == '{' and x == '}'):
                return False
    return stack = []
Queues support two operations:
- Enqueue
- Dequeue
- Retrieval from queues is first-in, first-out (FIFO)

Queues can be easily implemented by either arrays or linked lists.

Applications: Breadth first search, CPU scheduling, resource is shared among multiple consumers.
Sets

- A set contains a number of elements, with no duplicates and no order

Examples

- \( A = \{ 1, 5, 3, 96 \} \)
- \( B = \{ 17, 5, 1, 96 \} \)
- \( C = \{ \text{"Mary"}, \text{"contrary"}, \text{"quite"} \} \)
- Incorrect: \( \{ \text{"Mary"}, \text{"contrary"}, \text{"quite"}, \text{"Mary"} \} \)
Map

- Also known as *dictionary*, *associative array*
- Given two sets, Domain and Range, like a math function, each domain element has exactly one range element associated with it
- Two arrows can land on the same range element, but one domain element cannot have two arrows out of it
Mapping creates pairs of $<\text{DomainType}, \text{RangeType}>$

$<\text{key}, \text{value}>$ pairs

Basic operations

- put: add a key-value pair to a Map
- get: look up the value of a key
Map Example

'\text{A}' \rightarrow \text{“excellent”}

- Descriptions of grades: 'B' \rightarrow \text{“good”}
  'C' \rightarrow \text{“ok”}

- DomainType is char, and RangeType is string
- Each of these is a key-value pair, or just pair
- ('\text{A}', \text{“excellent”}) is a pair of the grade 'A' (key) and the phrase “excellent” (value)
- The whole mapping is the set of these 3 pairs
- \( M = \{ ('\text{A}', \text{“excellent”}), ('B', \text{“good”}), ('C', \text{“ok”}) \} \) – a map is a set of pairs, or “associations”
- Note that not every collection of pairs makes a proper map: \( M \) qualifies as a map only if the collection of keys has no duplicates