DS 221: Midterm Exam

Lokesh Mohanty (lokeshm@iisc.ac.in)

October 10, 2022

1. Set data structure

Design: Store in a sorted Array. Insert/delete by searching with binary search.
Justification: This design has $O(n)$ add, $O(log(n))$ exists and $O(n)$ delete complexities. It has fast read and check for existence. And also since the data is stored sorted, we can get the elements in sorted order in $O(n)$ time. Using linked list would have $O(n)$ add, $O(n)$ exists and $O(n)$ delete complexities. It would have slow read and check for existence. Hence I have used the sorted array design.

Algorithm 1: public Set(int capacity)

if `capacity` *is a non-negative integer* **then**
    int array[capacity]
    `this->data` $\leftarrow$ *array*
    `this->capacity` $\leftarrow$ *capacity*
else
    int array[1]
    `this->data` $\leftarrow$ *array*
    `this->capacity` $\leftarrow 1$
end
`this->size` $\leftarrow 0$

Steps:

1. Check if capacity is non-negative.
2. If true, initialize an array with the specified capacity.
3. Assign the location of the array to the class data property.
4. Set class property capacity and size.
5. In case of negative capacity, repeat the above steps for capacity 1.

Algorithm 2: public boolean add(int item)

```
first ← 0
last ← this->size
while first ≠ last do
    index ← (first+last)/2
    if this->data[index] > item then
        last ← index
    else if this->data[index] < item then
        first ← index + 1
    else
        return False
    end
end
if this->capacity = this->size then
    this->capacity ← this-capacity ×2
    int array[this->capacity]
    for i = 0 to first − 1 do
        array[i] ← this->data[i]
    end
    array[first] ← item
    for i = first + 1 to this->size do
        array[i] ← this->data[i-1]
    end
    this->data ← array
else
    for i =this->size to first + 1 do
        this->data[i] ← this->data[i-1]
    end
    this->data[first] ← item
end
return True
```

Steps:

1. Do a binary search to find the index of the item.
2. If found return False.
3. If not found, move all elements from the index to right by 1.
4. Store the new item at the index.
5. In case of max size, create a new array and move all elements there.

Algorithm 3: public boolean exists(int item)

```
first ← 0
last ← this->size
while first ≠ last do
    index ← (first+last)/2
    if this->data[index] > item then
        last ← index
    else if this->data[index] < item then
        first ← index + 1
    else
        return True
    end
end
return False
```

Steps:

1. Do a binary search to find the index of the item.
2. If found, move all elements after the index to left by 1.
3. If not found, return False

Algorithm 4: public boolean delete(int item)

```
first ← 0
last ← this->size
while first ≠ last do
    index ← (first+last)/2
    if this->data[index] > item then
        last ← index
    else if this->data[index] < item then
        first ← index + 1
    else
        this->size ← this->size −1
        for i = index to this->size−1 do
            this->data[i] ← this->data[i+1]
        end
        return True
    end
end
return False
```

Steps:

1. Do a binary search to find the index of the item.
2. If found, move all elements after the index to left by 1.
3. If not found, return False

2. Sort Stack

Algorithm: Use bubble sort. That is, pop the top element of the stack and compare with the next element and push the minimum into the temporary stack. Repeat this till the stack becomes empty. Then pop all elements from temporary stack and push into the main stack. This is same as one iteration of bubble sort. Repeat this till there are no swaps or n(length of stack) times.

Algorithm 5: void sort(stack st)

```
Initialize tempStack
numberOfSwaps ← 1
while numberOfSwaps ≠ 0 do
    numberOfSwaps ← 0
    top ← st.pop()
    while st.isEmpty() = False do
        if top < st.peek() then
            tempStack.push(top)
            top ← st.pop()
        else
            tempStack.push(st.pop())
            numberOfSwaps ← numberOfSwaps + 1
        end
    end
    while tempStack.isEmpty() = False do
        st.push(tempStack.pop())
    end
end
```

3. Adjacency Matrix vs Adjacency List

Adjacency List is good for sparse graphs (i.e., graphs with less number of edges but more vertices) as in this we only store the edges which exist.

Adjacency Matrix is good for dense graphs as we don't have to deal with the overhead of lists which become substantial for dense graphs and also reading takes O(max out degree) in case of adjacency list which become close to n in the case of dense graphs.

Algorithm 6: bool detectCycle(graph)

```
/* Assuming graph to be an adjacency list                                    */
Initialize Set notVisited
Initialize Queue toVisit
for i = 0 to graph.size() − 1 do
    /* Add all nodes to the notVisited set                                   */
    notVisited.add(graph[i])
end
while notVisited.size() > 0 do
    /* Add unconnected nodes                                                 */
    toVisit.enqueue(notVisited[0])
    parent ← null
    while toVisit is not empty do
        current ← toVisit.dequeue()
        notVisited.delete(current)
        for node in graph[current] do
            if node is not parent then
                if node is not in notVisited then
                    /* Cycle exists                                          */
                    return True
                else
                    toVisit.enqueue(node)
                end
            end
        end
        parent ← current
    end
end
/* No cycle found after traversing all the nodes                             */
return False
```

Steps:

- Store all non visited nodes and remove each one on visit.
- Until all nodes are visited, enqueue the first non visited node to a queue.
- Then remove it from non visited nodes and add all its neighbours other than its parent(the node which put it in the queue).
- If one of the nodes added by it is already visited then there is a cycle.
- Repeat this till a cycle is found or all the nodes are visited.
- No cycle exists if all the nodes are visited and no cycle is found.

4. Hash Table

(a) Best time complexity will be `O(1)` as when every key has a unique hash as when the index is known, finding item by index has `O(1)` time complexity in an array.

Worst time complexity will be `O(k)` as when every key has the same hash the lookup operation will traverse maximum `k` buckets to find the key.

Average/Expected time complexity will be `O(1)`. As we can see that the lookup operation traverses more than once if atleast 2 keys have the same hash and since its given that all the keys in the table are uniformly randomly distributed in $[0, R]$, we know that given that a particular bucket is filled, for another key to have the same hash, the probability is $\frac{\lfloor \frac{R}{b} \rfloor}{R} \approx \frac{1}{b}$. From this we can see as the number of traverses increases per lookup, its probability decreases exponentially. Since this converges to some constant we can say that it takes approximately constant time.

(b) Since Hash Table has very good lookup, it is good for storing dictionaries, implementing sets, and for all applications which require frequent lookups but less add and delete.

Hash Table doesn't support/benefit operations such as finding smallest/largest key, iterating on some range,.. Also it is bad for applications where there is high chance of data collision.

5. Complexity Analysis

$$
\begin{aligned}
f_1(n) &= 10^n && \implies log_2 f_1(n) = n log_2 10 \\
f_2(n) &= n^{1/3} && \implies log_2 f_2(n) = \frac{1}{3} log_2 n \\
f_3(n) &= n^n && \implies log_2 f_3(n) = n log_2 n \\
f_4(n) &= log_2 n && \implies log_2 f_4(n) = log_2 log_2 n \\
f_5(n) &= 2^{\sqrt{log_2 n}} && \implies log_2 f_5(n) = \sqrt{log_2 n}
\end{aligned}
$$

From this we can see that,

$$
\begin{aligned}
& O(log_2 f_4(n)) < O(log_2 f_5(n)) < O(log_2 f_2(n)) < O(log_2 f_1(n)) < O(log_2 f_3(n)) \\
\implies & O(f_4(n)) < O(f_5(n)) < O(f_2(n)) < O(f_1(n)) < O(f_3(n)),
\end{aligned}
$$

$$
\boxed{\therefore \text{growth rate: } f_4(n) < f_5(n) < f_2(n) < f_1(n) < f_3(n)}
$$

6. Complexity Analysis vs Empirical Analysis

Complexity Analysis

- Complexity Analysis is done for an algorithm.
- It is independent of the machine/hardware and can be used to universally compare algorithms.
- It allows to analyse the algorithm for all possible input.
- It can be used as a filter for most of the algorithms as it is not possible to do empirical analysis for all possible algorithms.

Empirical Analysis

- Empirical Analysis is done for an implementation of an algorithm
- In practice, we will often need to resort to empirical rather than complexity analysis to compare programs.
- It tells us about performance of the algorithm "on average" for real instances.
- The algorithm may not capture important effects of the hardware architecture that arise in practice.
- There may be implementational details that affect constant factors and are not captured by asymptotic analysis.
- But since it is machine dependent, it cannot be used to compare algorithms universally.
- The performance analysis may not be accurate as it depends on the data used. And it is generally impossible to check for all possible data.

7. In Order traversal without recursion

Algorithm 7: void inorderTraversal(root)

```
Initialize Stack st
top ← root
while top is not null do
    /* add root to stack if left exists                                          */
    while top->left exists do
        st.push(top)
        top ← top->left
    end
    /* print root if left doesn't exist                                          */
    print top->data
    /* pop from stack and print if right doesn't exist and stack is not empty    */
    while top->right doesn't exist and st.isEmpty() = False do
        top ← stack.pop()
        print top->data
    end
    /* run the loop for right sub-tree if it exists                              */
    top ← top->right
end
```

Steps:

1. Set root as the current node.
2. If left node exists, make it current and add root to stack.
3. Print the current node if left node doesn't exist.
4. If right node doesn't exist, pop the stack, set it as current node and print it.
5. Repeat the above step until the current node has a right node or the stack is not empty.
6. If right node exists for a node, goto step 1 with the right node as root.

8. PySpark

Pseudo Code:

```
marksRDD = sc.textFile(\marks.csv");
courseRDD = marksRDD.filter(lambda x : x.split(\,")[1] == \DS221");
pairRDD = courseRDD.map(lambda y :
( y.split(\,")[0], float(y.split(\,")[2]) ) );
gradesRDD = pairRDD.map(lambda k,v:
  (k, \A") if v > 80 else
    ((k, \B") if v > 70 else
       ((k, \C") if v > 60 else (k, \D"))
    )
  );

gradesRDD.collect();
```

Example:

```
1,DS221,90
2,DS211,85
3,DS221,50
4,DS211,40
5,DS221,70
6,DS220,72
7,DS221,65
8,DS220,69
9,DS221,75
```

- **marksRDD** contains an array of "studentID,courseID,marks"

  ```
  ["1,DS221,90",
  "2,DS211,85",
  "3,DS221,50",
  "4,DS211,40",
  "5,DS221,70",
  "6,DS220,72",
  "7,DS221,65",
  "8,DS220,69",
  "9,DS221,75"]
  ```

- **courseRDD** contains an array of "studentID,courseID,marks" with courseID as "DS221"

  ```
  ["1,DS221,90",
  "3,DS221,50",
  "5,DS221,70",
  "7,DS221,65",
  "9,DS221,75"]
  ```

- **pairRDD** contains an array of "(studentID,marks)" of course "DS221"

  ```
  [("1",90),
  ("3",50),
  ("5",70),
  ("7",65),
  ("9",75)]
  ```

- **gradesRDD** contains an array of "studentID,grades" based on the their marks

```
[("1","A"),
("3","D"),
("5","C"),
("7","C"),
("9","B")]
```

- `gradesRDD.collect()` displays the contents of `gradesRDD`

Output

```
[("1","A"),
("3","D"),
("5","C"),
("7","C"),
("9","B")]
```

9. Arrange 8 queens in a chess board

Algorithm: Start placing the queens from left to right. The first queen has 8 chances. Then next queen (in the second column) has 8 - places prohibited by 1st. The third queen has 8 - places prohibited by 1st and 2nd and so on.. Once all the 8 queens are placed, the list of queens is printed.

Algorithm 8: void arrangeQueens(queens)

```
currentColumn ← queens.size()
if currentColumn = 8 then
    print queens
else
    Initialize Set set
    for i = 0 to currentColumn − 1 do
        set.add(queens[i])
        if queens[i] + currentColumn − i ≤ 7 then
            set.add(queens[i] + currentColumn − i)
        end
        if queens[i] − currentColumn + i ≥ 0 then
            set.add(queens[i] − currentColumn + i)
        end
    end
    for i = 0 to 7 do
        if i doesn't exist in set then
            arrangeQueens(queens.push(i))
        end
    end
end
```