CS 261-020
Data Structures

Lecture 17
Additional Topics
3/8/22, Tuesday
Odds and Ends

• Assignment 5 rubrics will be posted by tonight

• Recitation 10
  • Go to your registered section
  • Cannot make up!
  • Implement a singly linked list and its basic functionalities using laptop provided
Assignment 5 Q&A
*Additional Topics

• Sets ADT and its implementation

• *Will not be on the final
Set

- **Set** – An ADT that can store *unique values*, *without any particular order*.
- **Unique** → no duplicates
- **Unordered** → cannot access items using index values

- Array: [1, 1, 2, 2, 3, 4, 1, 5, 8, 7]
- Set: {1, 2, 3, 4, 5, 8, 7} ← Note: no duplicates

- Why using set?
  - Check if a specific element is *contained* in the set
Set Operations

• The idea of a Set has been translated directly from mathematics into programming languages.
  • Such as in Python

• Basic operations:
  • `contains()` – search for a specific element and see if it is contained in the set
  • `add()` – add an element into the set
  • `remove()` – remove an element from the set
Set Operations

• More operations:
  • `union()` – return the union of two sets
  • Example:
    • A = \{2, 5, 7\}
    • B = \{1, 2, 5, 8\}
    • Then A Union B (A U B) = \{1, 2, 5, 8\}

• In Python:

```python
A = \{\texttt{\string'\textcolor{red}{red}', \string'\textcolor{green}{green}', \string'\textcolor{blue}{blue}'\}\}
B = \{\texttt{\string'\textcolor{yellow}{yellow}', \string'\textcolor{red}{red}', \string'\textcolor{orange}{orange}'\}\}
# by operator
print(A | B)
# by method
print(A.union(B))
```

Set Operations

• More operations:
  • `intersection()` – return the intersection of two sets
  • Example:
    • A = {2, 5, 7}
    • B = {1, 2, 5, 8}
    • Then A intersects B (A ∩ B) = {2, 5}

• In Python:

```python
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator
print(A & B)
# Prints {'red'}

# by method
print(A.intersection(B))
# Prints {'red'}
```
Set Operations

• More operations:
  • `difference()` – return the difference of two sets
  • Example:
    • \( A = \{2, 5, 7\} \)
    • \( B = \{1, 2, 5, 8\} \)
    • Then Set difference of \( A \) and \( B \) \( (A - B) = \{7\} \)

• In Python:
  ```python
  A = {'red', 'green', 'blue'}
  B = {'yellow', 'red', 'orange'}
  # by operator
  print(A - B)
  # Prints {'blue', 'green'}
  # by method
  print(A.difference(B))
  # Prints {'blue', 'green'}
  ```
Set Operations

• More operations:
  • `symmetric_difference()` – return the set of all elements in either A or B, but not both
  • Example:
    • A = \{2, 5, 7\}
    • B = \{1, 2, 5, 8\}
    • Then Set difference of A and B (A \^ B) = \{7, 1, 8\}

• In Python:

```python
A = \{'red', 'green', 'blue'\}
B = \{'yellow', 'red', 'orange'\}
# by operator
print(A - B)
# Prints \{'blue', 'green'\}
# by method
print(A.difference(B))
# Prints \{'blue', 'green'\}
```
Set Implementation

• Multiple ways of implementing a set ADT
  • Hash-based approach
  • Tree-based approach
Set Implementation: Using a Hash Table

• The underlying data structure is a hash table
  Key (element) $\rightarrow$ Hash Function $\rightarrow$ Index

• Use either chaining or open addressing to resolve collisions
Set Implementation: Using a Hash Table

• `contains()` – search for an element and see if it is contained in the set

• Similar to the `lookup()` in the hash table:
  • Take the element (key)
  • Apply the hash function, and get the index
  • Access

• Complexity: $O(1)$
Set Implementation: Using a Hash Table

• *add()* – add an element into the set

• Similar to the insert() in the hash table:
  • Take the element (key)
  • Apply the hash function, and get the index
  • Insert
    • Resize and rehash if needed
    • Resolve collision if needed

• Complexity: avg. O(1)
Set Implementation: Using a Hash Table

• `remove()` – remove an element from the set

• Similar to the `remove()` in the hash table:
  • Take the element (key)
  • Apply the hash function, and get the index
  • Remove
    • Add dummy node (tombstone) if needed

• Complexity: $O(1)$
Set Implementation: Using a Hash Table

• \textit{union(set A, set B)} – return the union of two sets

• Procedure:
  • Create an empty set, say S
  • Add all elements of A into S
  • Add all elements of B into S
  • Return S
  • *Note: since hash table cannot have duplicate keys, it handles “no duplicates” rule in Sets

• Complexity: \(O(\text{size}(A) + \text{size}(B))\)
Set Implementation: Using a Hash Table

• intersection(set A, set B) – return the intersection of two sets

• Procedure:
  • Create an empty set, say S
  • Loop through each element $A_i$ in set A
    • If $A_i$ is in B (by calling contains())
      • Add $A_i$ into S
  • Return S

• Complexity: $O(\min(\text{size}(A), \text{size}(B)))$
Set Implementation: Using a Hash Table

• *difference*(set A, set B) – return the difference of two sets
  • in this case: A - B

• Procedure:
  • Create an empty set, say S
  • Loop through each element Aᵢ in set A
    • If Aᵢ is NOT in B (by calling contains())
      • Add Aᵢ into S
  • Return S

• Complexity: \( O(\text{size}(A)) \)
Set Implementation: Using a Hash Table

- *symmetric_difference(set A, set B)* – return the symmetric difference of two sets

- Procedure:
  - Create an empty set, say S
  - Loop through each element $A_i$ in set A
    - If $A_i$ is NOT in B (by calling contains())
    - Add $A_i$ into S
  - Loop through each element $B_i$ in set B
    - If $B_i$ is NOT in A (by calling contains())
    - Add $B_i$ into S
  - Return S

- Complexity: $O(size(A)+size(B))$
Set Implementation: Using a Hash Table

• Example Set Implementation in C using hash table:
  • https://github.com/barrust/set
Set Implementation: Using a Tree

• The underlying data structure is a self-balancing tree:
  • AVL Tree
  • Red-black tree
Set Implementation: Using a Tree

• `contains()` – search for an element and see if it is contained in the set
• `add()` – add an element into the set
• `remove()` – remove an element from the set

• Similar to AVL tree’s `lookup()`, `insert()`, and `remove()`

• Complexity: $O(\log n)$ where $n$ is the number of element in the set
Set Implementation: Using a Tree

- **union(set A, set B)** – return the union of two sets

**Procedure:**
- Create an empty set S
- Insert all elements of A into S → n elements, each takes $O(\log n)$, so $O(n\log n)$
- For each element $B_i$ in B:
  - If S contains $B_i$, skip
  - Else, insert $B_i$ into S
- Return S
Set Implementation: Using a Tree

- **intersection(set A, set B)** – return the intersection of two sets

**Procedure:**
- Create an empty set, say S
- Loop through each element \( A_i \) in set A
  - If B contains \( A_i \)
    - Insert \( A_i \) into S
- Return S
Set Implementation: Using a Tree

• `difference(set A, set B)` – return the difference of two sets
  • in this case: A - B

• Procedure:
  • Create an empty set, say S
  • Loop through each element $A_i$ in set A
    • If $A_i$ is NOT in B (by calling `contains()`)  
    • Insert $A_i$ into S
  • Return S
Set Implementation: Using a Hash Table

• `symmetric_difference(set A, set B)` – return the symmetric difference of two sets

• Procedure:
  • Create an empty set, say S
  • Loop through each element $A_i$ in set A
    • If $A_i$ is NOT in B (by calling `contains()`)  
      • Insert $A_i$ into S
  • Loop through each element $B_i$ in set B
    • If $B_i$ is NOT in A (by calling `contains()`)  
      • Insert $B_i$ into S
  • Return S
Red-Black Tree

• Another type of self-balancing tree:

• Explore 4 YouTube videos here: