CS 261-020
Data Structures

Lecture 6
Complexity Analysis
Stack, Queue, Deque
2/1/24, Thursday
Odds and Ends

• Assignment 2 posted, due 2/11

• Due: Sunday 2/4 midnight
  • Quiz 2 (unlock after today’s lecture)
Lecture Topics:

• Complexity Analysis
  • Array Insertion
  • List insertion & removal

• Stacks, Queues, and Deques
  • Linear ADTs
Calculating time from Big O

• Ex. Merge sort, which is an $O(n \log n)$ algorithm, takes 96ms to sort an array of size 4000. Given this result, approximately how long merge sort will take to sort an array of size 1,000,000?

• Hint: $4000 \approx 2^{12}$, $1,000,000 \approx 2^{20}$

\[
\frac{f(n_1)}{f(n_2)} = \frac{t_1}{t_2} \Rightarrow \frac{n_1 \log n_1}{n_2 \log n_2} = \frac{t_1}{t_2}
\]

\[
\frac{2^{12} \log 2^{12}}{2^{20} \log 2^{20}} = \frac{96}{t_2}
\]

$\log_x x^y = y$
Worst case, Best case, and avg. case

• Note that the worst case, best case, and average case complexities of a data structure or an algorithm can differ, for example:

```c
int linear_search(int q, int* array, int n) {
    for (int i = 0; i < n; i++) {
        if (array[i] == q) {
            return i;
        }
    }
    return -1;
}
```

• Worst case: $O(n)$: if q appears to be the last element / does not exist
• Best case: $O(1)$: if q appears to be the first element
• Avg. case: $O(n)$: run about n/2 iterations, drop $\frac{1}{2}$
Real-world Consideration

• Your program will only perform as well as your design
  • Constant factors can still play a part

• Suppose you have two data structure or algorithms perform the same task:
  • A) 1,000,000n \(\rightarrow\) O(n)
  • B) 2 \(n^2\) \(\rightarrow\) O\((n^2)\)
  • Which one is better?
    • It depends

\[
\begin{align*}
\mathcal{f}_A(n) &= 1,000,000n \\
\mathcal{f}_B(n) &= 2n^2
\end{align*}
\]

\[n_D = \min\{ 0, 0, 0, 0 \} \]
Complexity of dynamic array insertion

• Recall: dynamic array insertion
  • Case 1: if size < capacity
    • Insert the new element
  • Case 2: if size == capacity
    • Step 1: allocate a new array that has twice the capacity
    • Step 2: copy all elements from data to new array
    • Step 3: delete the old data array and update data pointer
    • Step 4: Insert the new element

• Group Activity: What is the best-case, worst-case, and average case runtime complexities? Using Big O
Complexity of dynamic array insertion

• Group Activity: What is the best-case, worst-case, and average case runtime complexities?

• Best case: when \( \text{size} < \text{capacity} \)
  • Write the new value into the next open space
  • Time it takes to run this operation doesn’t depend on the size of the array \((n)\)
  • Thus, \(O(1)\)

• Worst case, when \( \text{size} \geq \text{capacity} \)
  • Require allocating a new array
  • Iterate through the \(n\) elements in the old array and copying them into the new array
  • Thus, \(O(n)\)
Complexity of dynamic array insertion

• Group Activity: What is the best-case, worst-case, and average case runtime complexities?

• How to determine average Case:
  • Use amortized analysis – a large cost is defrayed by spreading smaller payments over a period of time.
  • O(n) insertion cost (worst case) happens far less often than O(1) insertion cost (best case)
    • Since we double the capacity
  • Quantify the runtime complexity by aggregate analysis, by computing an upper bound $T$ on the total cost of a sequence of $n$ operations. Thus, average cost is $T / n$
Complexity of dynamic array insertion

• Assuming a dynamic array whose capacity starts at 1, doubled if resized. Perform a sequence of \( n \) insert. What’s the total cost?

• 1\textsuperscript{st} insertion: Write cost 1, copy cost 0
• 2\textsuperscript{nd} insertion: Write cost 1, copy cost 1 (resize)
• 3\textsuperscript{rd} insertion: Write cost 1, copy cost 2 (resize)
• 4\textsuperscript{th} insertion: Write cost 1, copy cost 0
• 5\textsuperscript{th} insertion: Write cost 1, copy cost 4 (resize)
• ......
Complexity of dynamic array insertion

- Assuming a dynamic array whose capacity starts at 1, doubled if resized. Perform a sequence of n insert. What’s the total cost?

- Create a table:

<table>
<thead>
<tr>
<th>Insertion # (resize # (k))</th>
<th>1</th>
<th>2 (1)</th>
<th>3 (2)</th>
<th>4</th>
<th>5 (3)</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9 (4)</th>
<th>10</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Copy cost</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>
\[
2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log n - 1} = \frac{1 \times (2^{\log n} - 1)}{2 - 1} = 2^n - 1
\]

**Complexity of dynamic array insertion**

<table>
<thead>
<tr>
<th>Insertion # (resize # (k))</th>
<th>1</th>
<th>2 (1)</th>
<th>3 (2)</th>
<th>4</th>
<th>5 (3)</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9 (4)</th>
<th>10</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>Copy cost</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

- Total **Write cost** = \(n\) \[\frac{1 \times n}{1} = 1 + 2 + 4 + 8 + 16 + \cdots + 2^n - 1\]
- Total **copy cost**: \[1 + 2 + 4 + 8 + 16 + \cdots + 2^{\log n - 1} = \frac{1 \times (2^{\log n} - 1)}{2 - 1} = 2^n - 1\]
Complexity of dynamic array insertion

<table>
<thead>
<tr>
<th>Insertion # (resize # ((k)))</th>
<th>1</th>
<th>2 (1)</th>
<th>3 (2)</th>
<th>4</th>
<th>5 (3)</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9 (4)</th>
<th>10</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>Copy cost</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

- Total cost = Total Write cost + Total copy cost:
  
  \[ \text{Total cost} = n + (n - 1) \]

  \[ \frac{2n - 1}{n} = 2 - \frac{1}{n} \]

  \[ \log_2 n = \log_{10} x \]

- Thus, average is \((2n-1)/n = O(1)\)
Complexity of dynamic array insertion

• Thus, average case is $(2n-1)/n = O(1)$

• On average, dynamic array insertion is a constant time operation.
Complexity of linked list insertion

• Assuming that we already know exactly where in the list we want to insert a new value (e.g. at the head or at the tail).

• Steps:
  • Allocating a new node
  • Updating pointers

• All run in constant time, thus, the runtime complexity is $O(1)$
  • For best, worst, and average cases
Complexity of linked list removal

• Assuming that we already know exactly where in the list we want to remove.

• Steps:
  • Updating pointers
  • Free the node

• All run in constant time, thus, the runtime complexity is $O(1)$
  • For best, worst, and average cases
### Dynamic Array vs. Linked List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Removal</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Access the nth element</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Lecture Topics:

• Complexity Analysis
  • Array Insertion
  • List insertion

• Stacks, Queues, and Deques
  • Linear ADTs
Stacks

- A linear ADT that imposes a Last In, First Out (LIFO) order on elements
  - The last element inserted must be the first one to remove
  - Real life examples: a stack of books, a stack of dishes, web browser’s “back” history, “undo” operation in a text editor

- A stack ADT has two ends: top and bottom
  - New elements can only be inserted at top
  - Only the element at the top may be removed

- Two main operations:
  - Push – inserts an element on the top
  - Pop – removes the top element
Implement Stack using Linked List

• Using a singly linked list, head of the list = the top of the stack

• When a value is **pushed** into a stack, it becomes the new head of the list

• When a value is **popped**, the current head of the list is removed
  • The **next node** becomes the new head
Implement Stack using Linked List

push(8)
pop()
Implement Stack using Linked List

• Complexity Analysis:
  • Push() – $O(1)$

  • Pop() – $O(1)$

*For all best-case, worst-case, and average-case
Implement Stack using Dynamic Array

- Using dynamic array, the end of the array = head of the stack

- When a new element is pushed onto the stack, it is inserted at the end of the array
  - Resize if needed, as a normal dynamic array

- When an element is popped, the array’s last element is removed
Implement Stack using Dynamic Array

Push (8)

Push (16)

Push (32)

Pop
Implement Stack using Dynamic Array

• Complexity Analysis
  • Pop() – $O(1)$
    • for all best-case, worst-case, and average case
  
  • Push()
    • $O(1)$ Best-case and average case
    • $O(n)$ worst-case (when resize is needed)
Queues

• A linear ADT that imposes a First In, First Out (FIFO) order on elements
  • The first element to be removed is the first one that was placed into it
  • Real life examples: a line of people waiting for check out

• A Queue ADT has two ends: front and back
  • Inserting elements to the back
  • Removing elements from the front

• Two main operations:
  • Enqueue – insert an element at the back
  • Dequeue – remove an element at the front
Queues
Implement Queue using Linked List

• Using a singly linked list. Must keep track of both the head and the tail of the list

• Enqueue onto the back → insert at the tail of the list

• Dequeue from the front → remove from the head of the list
Implement Queue using Linked List

enqueue(7)
dequeue()
Implement Queue using Linked List

• Complexity Analysis:
  • enqueue() – $O(1)$
  • dequeue() – $O(1)$

*for all best-case, worst-case, and average case
Implement Queue using Dynamic Array

• Using a dynamic array,
  • Front of the queue = front of the array
  • Back of the queue = back of the array

• Ex. A queue with 3 values (1 at the front, 5 at the back)

  0 1 2 3
  1 3 5

• Enqueue a new value → insert it at the end of the array

• What about dequeue?
Implement Queue using Dynamic Array

• Dequeue:
  • Option 1: remove the front, and shift all the remaining to left
    • Drawback: O(n) runtime complexity for each dequeue → NOT GOOD!!
  • Option 2: allow the front of the queue to “float” back into the middle of the array.
    • Need to keep track of the start of the data
Implement Queue using Dynamic Array

- enqueue(7)
- enqueue(9)
- dequeue()
Implement Queue using Dynamic Array

• An array that allows data to wrap around from the back to the front is known as a **circular buffer**

• Q: How do we know which index corresponds to the back of the queue?
  • By computing a mapping between the array’s **logical indices** and its **physical indices**

• Logical indices – the indices relative to the **start of the data**

• Physical indices – the indices relative to the **start of the physical array**
Implement Queue using Dynamic Array

• Mapping formula: \( \text{physical} = \text{start} + \text{logical} \);

• Since it is circular, add the following to check:
  
  ```java
  if (\text{physical} >= \text{capacity}) {
      \text{physical} -= \text{capacity};
  }
  ```

  OR: \( \text{physical} = (\text{start} + \text{logical}) \mod \text{capacity} \);

• Index at which the next element will be inserted:
  • Previously: \( \text{array}[\text{size}] \) – when the data starts at physical index 0
  • Now: \( \text{array}[\text{physical}] \) – where physical = (start + size) \mod capacity
Implement Queue using Dynamic Array

• Dynamic Array resizing for the queue implementation
• When do we need to resize?
  • size >= capacity
• When resize, reindex!
  • Logical index 0 ↔ Physical index 0

• How?
  • Loop through the logical indices from 0 to size – 1
  • Copy elements at each logical index in the old array to the equivalent physical index in the new array
Implement Queue using Dynamic Array

• Visually, look like this:
Implement Queue using Dynamic Array

• Complexity:
  • Dequeue – $O(1)$ for all best-case, worst-case, and average case
  
  • Enqueue
    • $O(1)$ for best-case and average case
    • $O(n)$ for worst-case, when resize is needed