Arrays and Linked Lists

Josh Wilcox (jw14g24@soton.ac.uk)

February 4, 2025

Contents

1 Arrays

1.1 Fixed Length

1.1.1 Properties

- Contiguous chunk of memory
  - All the elements of the array are stored in adjacent memory locations, one right after the other, without any gaps in between
- Has an access time of $\Theta(1)$
- Very efficient use of memory
  - Often provides best performance

1.1.2 Disadvantages

- Fixed length - Can be variable at extra time cost
- Insertion and deletion to/from the middle have $\Theta(n)$ time complexity

1.2 Variable Length

- Most `add(elem)` operations are $\Theta(1)$
- When a chunk of memory for an array is full, we need to copy all elements to a new larger chunk in order to add more elements
  - This resizing operation has a time complexity of $\Theta(n)$, where n is the number of elements in the array

1.3 General Time Analysis of Adding Elements

- To perform N adds with an initial capacity of n
- We must perform m copies where:

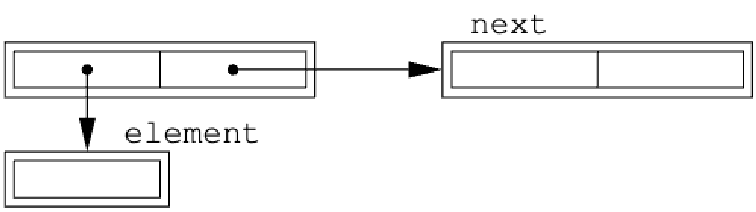
$$n \cdot 2^{m-1} < N \leq n \cdot 2^m$$

- Total Elements copied

$$\sum_{i=1}^{m} 2^{i-1} \cdot n = n(2^m - 1)$$

2 Linked Lists

- Non-Contiguous data use pointers to reference their units of data
- Removes the disadvantage of Contiguous data structures of adding and removing data from the middle being expensive
- A linked list is built up of nodes
- Each node contains a value and a reference to the next node in the list
- The final value in the list has a null reference



2.1 Singly Linked List

- A linked list that simply is a string of nodes to eachother in **one direction**
- Has a single pointer to the **next node only**
- Differs to a doubly linked lists which has pointers to the *next and previous* nodes

2.1.1 Java Implementation

```java
/**
 * A simple implementation of a singly linked list in Java.
 * This class demonstrates the basic operations of a linked list.
 */
public class MyLinkedList<E> {
    // The head node of the linked list
    private Node<E> head;
    // The number of elements in the linked list
    private int noElements;

    /**
     * A static nested class representing a node in the linked list.
     * Each node contains an element and a reference to the next node.
     */
    private static class Node<T> {
        private T element; // The data stored in the node
        private Node<T> next; // Reference to the next node in the list
    }

    /**
     * Constructor to create an empty linked list.
     * Initially, the head is null and the number of elements is zero.
     */
    public MyLinkedList() {
        head = null;
        noElements = 0;
    }

    /**
     * Returns the number of elements in the linked list.
     * @return the size of the linked list
     */
    public int size() {
        return noElements;
    }

    /**
     * Checks if the linked list is empty.
     * @return true if the linked list is empty, false otherwise
     */
    public boolean isEmpty() {
```

```java
        return head == null;
    }

    /**
     * Adds a new element to the front of the linked list.
     * @param element the element to be added
     * @return true if the element is added successfully
     */
    public boolean add(E element) {
        // Create a new node with the given element
        Node<E> newNode = new Node<E>();
        newNode.element = element; // Set the element of the new node
        newNode.next = head; // Set the next reference of the new node to the
        ↪  current head
        head = newNode; // Update the head to the new node
        noElements++; // Increment the number of elements
        return true;
    }

    /**
     * Removes the head (first element) of the linked list.
     * @return true if the head is removed successfully, false if the list is empty
     */
    public boolean remove_head() {
        if (!isEmpty()) {
            head = head.next; // Update the head to the next node
            noElements--; // Decrement the number of elements
            return true;
        }
        return false; // Return false if the list is empty
    }

    /**
     * Checks if the linked list contains a specific element.
     * @param obj the element to check for
     * @return true if the element is found, false otherwise
     */
    public boolean contains(E obj) {
        for (Node<E> current = head; current != null; current = current.next) {
            if (obj.equals(current.element)) {
                return true;
            }
        }
        return false;
    }
}
```

3 Stack Implementation with Linked Lists

```
public class LinkedListStack<E>
{
    private MyLinkedList<E> list = new MyLinkedList<E>();

    boolean push(E obj) { // New element becomes the head
        return list.add(obj);
        }

    E peek() { // Only return the head of the linked list
        return list.get_head();
        }

    E pop() { // Removes the head of the linked list
        if (isEmpty()) throw EmptyStackException;
        T elem=list.get_head();
        list.remove_head();
        return elem;
    }
    boolean isEmpty() {
        return list.isEmpty();
        }
}
```

- Stack operations of linked list take constant time - $\Theta(1)$
- There is a hidden cost of creating and destroying `Node` objects
- Memory requirement is $\Theta(n)$
- Array implementation of stacks are better in practice - constant time with no hidden cost

4 Queue Implementation with Linked List

- For Queues, we add at one end and remove from the other.
- Use a **Head and Tail**
  - Enqueue at the back
  - Dequeue at the head

5 Java Linked Lists

- Allows add and remove at **both** ends of the list
- Uses a doubly linked list with pointers to the next and previous nodes
- Uses a new node that has a null value but `next` and `previous` reference the head and tail. Last element references this node.

6 Skip Lists

6.1 Introduction

- Skip lists are a data structure that allows fast search within an ordered sequence of elements.
- They are a probabilistic alternative to balanced trees.
- Skip lists use multiple levels of linked lists to achieve logarithmic time complexity for search, insertion, and deletion operations.

6.2 Structure

- A skip list consists of multiple layers, where each layer is a sorted linked list.
- The bottom layer contains all the elements, and each higher layer acts as an "express lane" with fewer elements.
- Each element in a layer has a reference to the next element in the same layer and possibly to an element in the layer below.

6.3 Operations

6.3.1 Search

- Start from the top-left element.
- Move right until the next element is greater than or equal to the target.
- Move down one level and repeat until the target is found or the bottom level is reached.
- Average time complexity: $O(\log n)$.

6.3.2 Insertion

- Perform a search to find the position where the new element should be inserted.
- Insert the element in the bottom layer.
- Promote the element to higher layers with a certain probability (e.g., 50% chance).
- Average time complexity: $O(\log n)$.

6.3.3 Deletion

- Perform a search to find the element to be deleted.
- Remove the element from all layers where it appears.
- Average time complexity: $O(\log n)$.

6.4 Advantages and Disadvantages

- **Advantages:**
  - Simple to implement.
  - Provides fast search, insertion, and deletion operations.
  - Probabilistic balancing without the need for complex rotations.
- **Disadvantages:**
  - Performance can degrade if the randomization does not work well.
  - Requires additional memory for multiple levels of pointers.