Chapter 10. Elementary Data Structures

Joon Soo Yoo

May 1, 2025

Assignment

- Read §10
- Problems
  - §10.1 - 2, 4, 5
  - §10.2 - 1, 2, 3
  - §10.3 - 1, 2, 4

Part III: Data Structures

- Chapter 10: Elementary Data Structures
- Chapter 11: Hash Tables
- Chapter 12: Binary Search Trees
- Chapter 13: Red-Black Trees

Operations on Dynamic Sets

Dynamic set can *grow, shrink, or change* over time.

Dynamic sets support two types of operations:

- **Queries** – return information without modifying the set
- **Modifications** – alter the contents of the set

Typical operations include:

- `SEARCH(S, k)` – returns pointer to element with key k or `NIL`
- `INSERT(S, x)` – inserts element x into set S
- `DELETE(S, x)` – deletes element pointed to by x from S

More Dynamic Set Operations

On totally ordered sets, we also define:

- `MINIMUM(S)` – returns pointer to element with smallest key
- `MAXIMUM(S)` – returns pointer to element with largest key
- `SUCCESSOR(S, x)` – next element after x in sorted order
- `PREDECESSOR(S, x)` – previous element before x in sorted order

Chapter 10: Elementary Data Structures

- **Chapter 10.1: Simple Array-Based Structures (Arrays, Matrices, Stacks, Queues)**
- Chapter 10.2: Linked Lists
- Chapter 10.3: Representing Rooted Trees

Arrays: Memory Layout

Memory layout: Arrays are allocated as a contiguous block of memory with a fixed stride (i.e., same number of bytes per element).
Index-based access: To access $A[i]$, the system computes the address using:

$$\text{address} = a + i \cdot b$$

where:

- a: base address
- b: fixed byte size per element

Matrix Representation Overview

Matrix: $m \times n$ matrix M

- Represented as a 2D array using 1D arrays
- Two common linear storage schemes:
  - **Row-major:** store row-by-row
  - **Column-major:** store column-by-column

Row-Major vs Column-Major Order

Let $M[i][j]$ be the element at row i and column j.

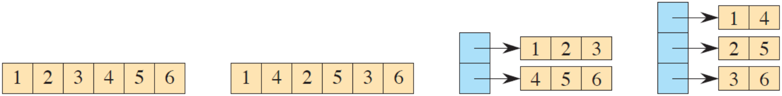
- **Row-major:** element index $= n \cdot i + j$
- **Column-major:** element index $= i + m \cdot j$

Example Matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Row-major: $[1, 2, 3, 4, 5, 6]$
- Column-major: $[1, 4, 2, 5, 3, 6]$

Multi-Array Representations



(a) (b) (c) (d)

Row Pointers:

- One array for each row
- Master array holds pointers to row arrays
- Access $M[i][j]$ via $A[i][j]$

Column Pointers:

- One array for each column
- Master array holds pointers to column arrays
- Access $M[i][j]$ via $A[j][i]$

Block Representation

Block Storage:

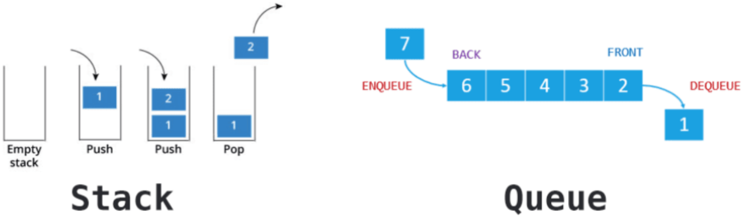
- Divide the matrix into blocks (e.g., 2×2 blocks)
- Store blocks contiguously in memory

Example: 4×4 **matrix stored in** 2×2 **blocks:**

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Block Order: $[1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16]$

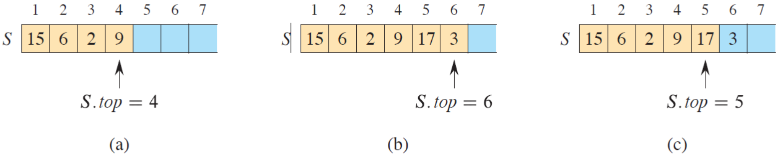
Stacks: Overview and Intuition



Stack Basics

- Stack = dynamic set with **LIFO** (Last-In, First-Out) behavior
- `INSERT` → `PUSH`, `DELETE` → `POP`
- Analogy: Cafeteria plate dispenser
- Most recently inserted element is the first to be removed

Stacks: Array Implementation



- Use array $S[1..n]$ to hold stack elements
- Attributes:
  - $S.\texttt{top}$: index of most recently inserted element
  - $S.\texttt{size}$: capacity of the stack (i.e., n)
- Stack consists of elements $S[1..S.\texttt{top}]$

Stack Procedures

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE

PUSH(S, x)
if S.top == S.size
    error "overflow"
else S.top = S.top + 1
    S[S.top] = x

POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Stack Operations and Overflow/Underflow

Special Conditions

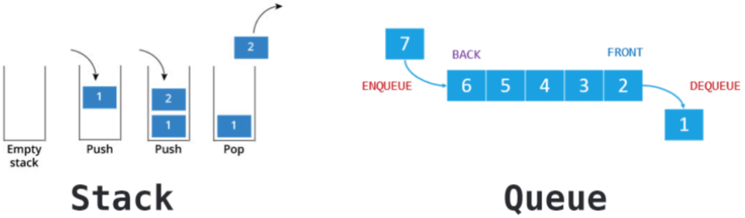
- $S.\texttt{top} = 0 \Rightarrow$ Stack is empty
- $S.\texttt{top} = S.\texttt{size} \Rightarrow$ Stack is full

Error handling:

- `POP` on empty stack: underflow error
- `PUSH` when full: overflow error

All operations run in $\Theta(1)$ time.

Queues: Overview

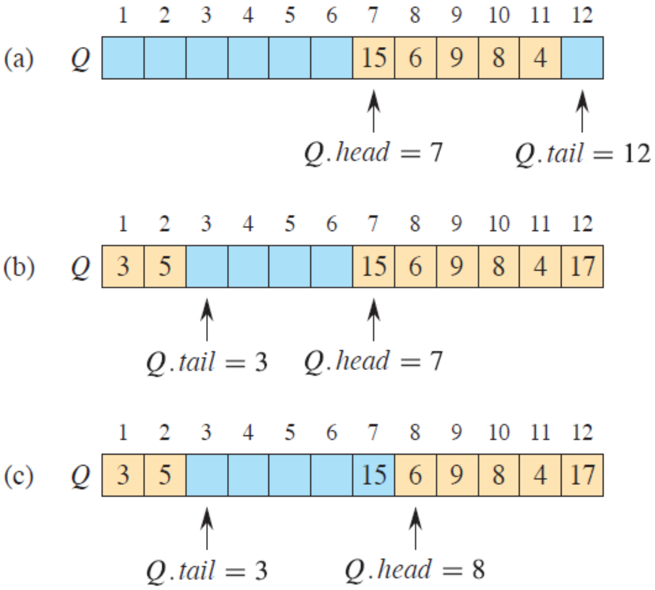


- **ENQUEUE**: Insert operation (adds to tail)
- **DEQUEUE**: Delete operation (removes from head)
- Queue follows **FIFO** (First-In, First-Out) policy
- Similar to a line of customers: new arrivals go to the back, service from the front

Queues: Array-Based Implementation

- Use array $Q[1 \ldots n]$ to store queue elements
- Attributes:
  - $Q.size$: total size of the array (capacity n)
  - $Q.head$: index of the front (dequeue from here)
  - $Q.tail$: index of the next insertion (enqueue to here)
- Queue elements stored from $Q.head$ to $Q.tail - 1$
- Indices **wrap around** circularly: index 1 follows n
- We store at most $n - 1$ elements to distinguish full vs. empty:
  - **Empty queue:** $Q.head = Q.tail$
  - **Full queue:** $Q.head = Q.tail + 1$

Queues: Example



Queue Operations

ENQUEUE(Q, x)

```
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else Q.head = Q.head + 1
return x
```

In the procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow.

Chapter 10: Elementary Data Structures

- Chapter 10.1: Simple Array-Based Structures (Arrays, Matrices, Stacks, Queues)
- **Chapter 10.2: Linked Lists**
- Chapter 10.3: Representing Rooted Trees

Linked Lists: Definition

A **linked list** is a data structure in which elements are arranged in a linear order, but *unlike arrays*, the order is determined by pointers rather than indices.

- Each element (node) contains a pointer to the next element in the list
- This structure supports dynamic insertion and deletion
- Because nodes may contain searchable keys, linked lists are also known as **search lists**

Doubly Linked List: Structure

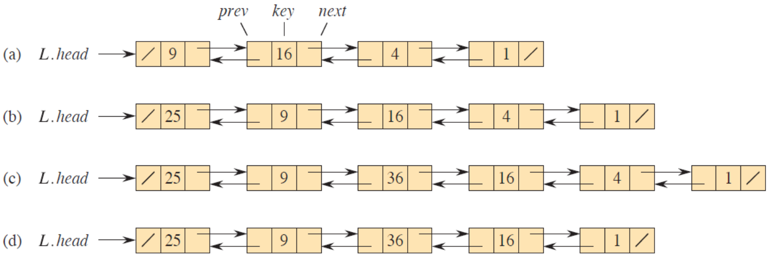
In a **doubly linked list** L, each element is an object with:

- `key` – the data or identifier
- `next` – pointer to the successor
- `prev` – pointer to the predecessor

Special cases:

- If $x.\mathtt{prev} = \mathtt{NIL}$, then x is the **head** (first element)
- If $x.\mathtt{next} = \mathtt{NIL}$, then x is the **tail** (last element)
- The list itself has attribute $L.\mathtt{head}$ pointing to the first element
- If $L.\mathtt{head} = \mathtt{NIL}$, the list is **empty**

Doubly linked list



Variants of Linked Lists

A linked list may have different structural properties:

- **Singly vs. Doubly Linked**
  - *Singly linked:* only `next` pointers
  - *Doubly linked:* both `next` and `prev` pointers
- **Sorted vs. Unsorted**
  - *Sorted:* keys are in increasing order; `head` is minimum, `tail` is maximum
  - *Unsorted:* elements can appear in any order
- **Circular vs. Linear**
  - *Circular:* `head.prev` points to `tail`, and `tail.next` points to `head`
  - *Linear:* first and last elements have `NIL` in one direction

Unless stated otherwise, we assume lists are doubly linked and unsorted.

Searching a Linked List

```
LIST-SEARCH(L, k)
    x = L.head
    while x ≠ NIL and x.key ≠ k
        x = x.next
    return x
```

LIST-SEARCH(L, k) performs a linear search to find the first element in list L with key k.

- Scans the list from head to tail
- Returns a pointer to the first element with key k
- If no such element exists, returns `NIL`

Worst-case time: $\Theta(n)$, when the key is at the end or not present at all.

Prepending into a Linked List

LIST-PREPEND(L, x) inserts element x at the front of list L.

```
LIST-PREPEND(L, x)
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

- **Time:** $\mathcal{O}(1)$

Inserting into a Linked List

LIST-INSERT(y, x**)** inserts x *immediately after* an existing element y in the list.

```
LIST-INSERT(x, y)
    x.next = y.next
    x.prev = y
    if y.next ≠ NIL
        y.next.prev = x
    y.next = x
```

- **Time:** $\mathcal{O}(1)$

Deleting from a Linked List

LIST-DELETE(x) removes an element x from a linked list by *splicing it out*.

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
        x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
        x.next.prev = x.prev
```

- Takes $\mathcal{O}(1)$ time

Linked Lists vs. Arrays: Time Trade-offs

Insertion and Deletion:

- **Doubly Linked List:** $\mathcal{O}(1)$ time (just update pointers)
- **Array:** $\Theta(n)$ time if shifting elements to maintain order in the worst case

Accessing the kth element:

- **Array:** $\mathcal{O}(1)$ time (direct indexing)
- **Linked List:** $\Theta(k)$ time (must traverse k nodes)

Conclusion: Use **linked lists** for efficient updates, use **arrays** for fast indexed access.

Arrays vs. Linked Lists: Summary of Trade-offs

Operation	**Array**	**LL (w/ pointer)**	**LL (w/o pointer)**
Insert at front	$\Theta(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert after a node	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$
Delete a given node	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$
Access kth element	$\mathcal{O}(1)$	$\Theta(k)$	$\Theta(k)$

Key point: Linked lists are efficient when you already have a pointer; arrays are better for random access.

Sentinels in Doubly Linked Lists

Sentinel: A special dummy object $L.\texttt{nil}$ used to simplify boundary cases in a linked list.

Circular, doubly linked list with sentinel:

- $L.\texttt{nil.next}$ points to the **head**
- $L.\texttt{nil.prev}$ points to the **tail**
- The list is circular: head's `prev` and tail's `next` point back to $L.\texttt{nil}$
- Removes the need for special cases at the head or tail

Note: Do not delete the sentinel $L.\mathit{nil}$ unless destroying the entire list.

Sentinels in Doubly Linked Lists (Diagram)

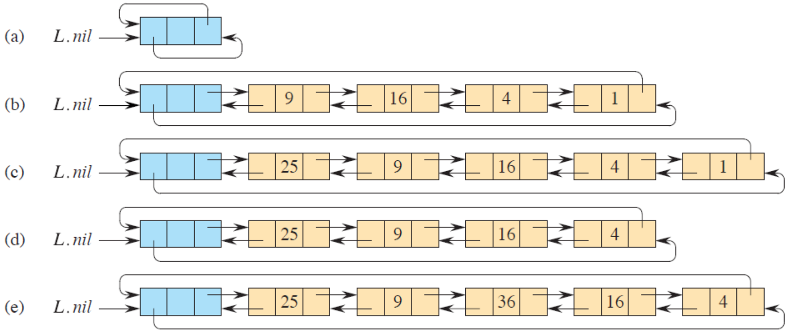


Figure 10.5 A circular, doubly linked list with a sentinel. The sentinel $L.nil$, in blue, appears between the head and tail. The attribute $L.head$ is no longer needed, since the head of the list is $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.4(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(x, L.nil)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4. **(e)** The list after executing LIST-INSERT$'(x, y)$, where $x.key = 36$ and y points to the object with key 9.

LIST-DELETE’ using Sentinel

```
LIST-DELETE'(x)
x.prev.next = x.next
x.next.prev = x.prev
```

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

LIST-INSERT’ using Sentinel

```
LIST-INSERT'(x, y)
x.next = y.next
x.prev = y
y.next.prev = x
y.next = x
```

```
LIST-INSERT(x, y)
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

Searching with a Sentinel

Searching a circular, doubly linked list with a sentinel has the same asymptotic cost ($\Theta(n)$) as without one — but it can reduce the **constant factor** in practice.

Why?

- Normal search checks:
  - Is $x = \texttt{NIL}$?
  - Does $x.\texttt{key} = k$?
- With a sentinel, you *guarantee* the key will be found (either in the list or in the sentinel)
- This eliminates the end-of-list check in each iteration

Sentinels simplify logic and reduce comparisons, but use memory. In this book, we use sentinels only when they significantly simplify the code.

Searching with a Sentinel

```
LIST-SEARCH'(L, k)
L.nil.key = k            // store the key in the sentinel to guarantee it is in list
x = L.nil.next           // start at the head of the list
while x.key ≠ k
    x = x.next
if x == L.nil            // found k in the sentinel
    return NIL           // k was not really in the list
else return x            // found k in element x
```

Searching without a Sentinel

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

Chapter 10: Elementary Data Structures

- Chapter 10.1: Simple Array-Based Structures (Arrays, Matrices, Stacks, Queues)
- Chapter 10.2: Linked Lists
- **Chapter 10.3: Representing Rooted Trees**

Rooted Tree Representation

Motivation: Linked lists work well for linear data, but many relationships (e.g., hierarchies) are *nonlinear*.

In this section:

- We study how to represent **rooted trees** using linked structures
- We start with **binary trees**, then extend to trees with arbitrary branching

Each node is represented as an object with:

- A `key` attribute (like linked lists)
- Additional pointers that vary by tree type

Binary Tree Representation

In a **binary tree**, each node x contains:

- $x.\mathtt{p}$ – pointer to the parent
- $x.\mathtt{left}$ – pointer to the left child
- $x.\mathtt{right}$ – pointer to the right child

Special cases:

- If $x.\mathtt{p} = \mathtt{NIL}$, then x is the **root**
- If $x.\mathtt{left} = \mathtt{NIL}$, x has no left child
- If $x.\mathtt{right} = \mathtt{NIL}$, x has no right child

The entire tree T is accessed via the pointer $T.\mathtt{root}$. If $T.\mathtt{root} = \mathtt{NIL}$, the tree is **empty**.

Binary Tree Diagram

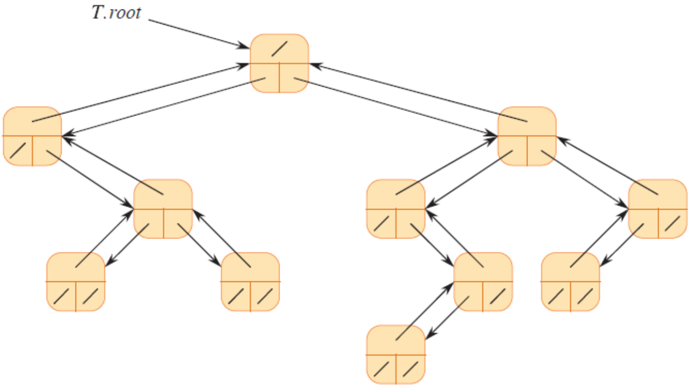


Figure 10.6 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

Unbounded Branching: The Problem

What if a node has many children?

Naive approach:

- Use attributes `child1`, `child2`, ..., `child`k for each node
- Works only when the number of children $\leq k$ and k is small

Problems:

- Not flexible when number of children is unbounded or varies
- Wastes memory if most nodes have few children but k is large

We need a more space-efficient, general-purpose representation.

Left-Child, Right-Sibling Representation

To handle arbitrary branching with $\mathcal{O}(n)$ space, we use the:
Left-child, Right-sibling representation

Each node x stores:

- $x.\texttt{p}$: pointer to parent
- $x.\texttt{left-child}$: pointer to leftmost child
- $x.\texttt{right-sibling}$: pointer to immediate sibling to the right

Special cases:

- If x has no children, $x.\texttt{left-child} = \text{NIL}$
- If x is the last child, $x.\texttt{right-sibling} = \text{NIL}$

This structure generalizes binary trees and supports arbitrary tree shapes.

Left-Child, Right-Sibling Diagram Example

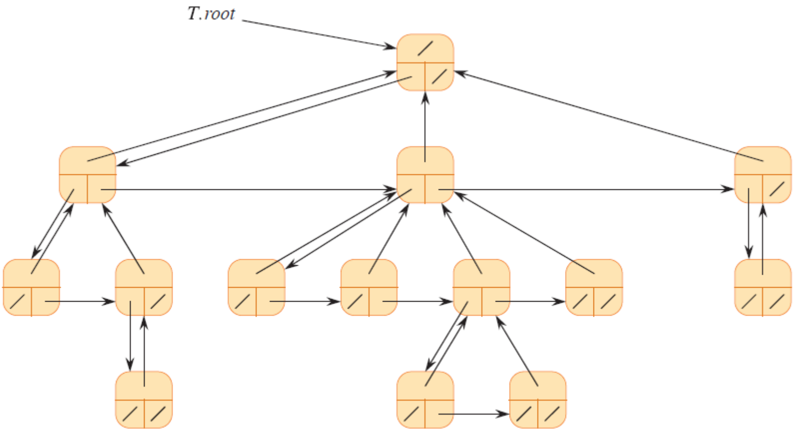


Figure 10.7 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.\textit{left-child}$ (lower left), and $x.\textit{right-sibling}$ (lower right). The *key* attributes are not shown.

Other Tree Representations

Rooted trees can be represented in different ways depending on the application.

- **Heaps (Chapter 6):**
  - Represented as complete binary trees
  - Stored in a single array
  - Includes an attribute for the index of the last node
- **Chapter 19 Trees:**
  - Traversed only toward the root
  - Only parent pointers are present (no child pointers)

Many other representations are possible — the best one depends on the application.

Question?