Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Data Structures

- **Data structure.** Method for organizing data for efficient access, searching, manipulation, etc.
- **Goal.**
  - Fast.
  - Compact
- **Terminology.**
  - Abstract vs. concrete data structure.
  - Dynamic vs. static data structure.

Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays
Stack
- **Stack.** Maintain dynamic sequence (stack) S supporting the following operations:
  - `PUSH(x)`: add x to S.
  - `POP()`: remove and return the most recently added element in S.
  - `IS EMPTY()`: return true if S is empty.

![Stack Operations](image)

Queue
- **Queue.** Maintain dynamic sequence (queue) Q supporting the following operations:
  - `ENQUEUE(x)`: add x to Q.
  - `DEQUEUE()`: remove and return the first added element in Q.
  - `IS EMPTY()`: return true if S is empty.

![Queue Operations](image)

Applications
- **Stacks.**
  - Virtual machines
  - Parsing
  - Function calls
  - Backtracking
- **Queues.**
  - Scheduling processes
  - Buffering
  - Breadth-first searching

Stack Implementation
- **Stack.** Stack with capacity N
- **Data structure.**
  - Array S[0..N-1]
  - Index top. Initially top = -1
- **Operations.**
  - `PUSH(x)`: Add x at S[top+1], top = top + 1
  - `POP()`: return S[top], top = top - 1
  - `IS EMPTY()`: return true if top = -1.
  - Check for overflow and underflow in `PUSH` and `POP`.

![Stack Implementation](image)
Stack Implementation

N = 10

- Time
  - Push in Θ(1) time.
  - Pop in Θ(1) time.
  - isEmpty in Θ(1) time.
- Space
  - Θ(N) space.
- Limitations
  - Capacity must be known.
  - Wasting space.

Queue Implementation

N = 10

- Queue. Queue with capacity N.
- Data structure.
  - Array Q[0..N-1]
  - Indices head and tail and a counter.
- Operations.
  - enqueue(x): add x at S[tail], update count og tail cyclically.
  - dequeue(): return Q[head], update count og head cyclically.
  - isEmpty(): return true if count = 0.
  - Check for overflow and underflow in dequeue and enqueue.

Queue and Stacks

- Stack.
  - Time. Push, Pop, isEmpty in Θ(1) time.
  - Space. Θ(N)
- Queue.
  - Time. Enqueue, Dequeue, isEmpty in Θ(1) time.
  - Space. Θ(N)
- Challenge. Can we get linear space and constant time?
Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Linked Lists

- Doubly-linked lists in Java.

```java
class Node {
    int key;
    Node next;
    Node prev;
}
Node head = new Node();
Node b = new Node();
Node c = new Node();
head.key = 7;
b.key = 42;
c.key = 18;
head.prev = null;
head.next = b;
b.prev = head;
b.next = c;
c.prev = b;
c.next = null;
```

Linked Lists

- Simple operations.

```java
head
null 7 null 42 null 18 null
```

Linked Lists

- Linked lists.
  - Data structure to maintain *dynamic* sequence of elements in linear space.
  - Sequence order determined by pointers/references called *links*.
  - Fast insertion and deletion of elements and contiguous sublists.
  - *Singly-linked* vs *doubly-linked*.

```
Linked Lists

head
null 7 42 18 23 5 null
```

Linked Lists

- Simple operations.

```java
search(head, k): return node with key k. Return null if it does not exist.
insert(head, x): insert node x in front of list. Return new head.
delete(head, x): delete node x in list.
```

```
Linked Lists

head
null 7 42 18 23 5 null
```

Linked Lists

- Simple operations.

```java
search(head, k): return node with key k. Return null if it does not exist.
insert(head, x): insert node x in front of list. Return new head.
delete(head, x): delete node x in list.
```

```
Linked Lists

head
null 7 42 18 23 5 null
```
Ex. Let p be a new with key 10 and let q be node with key 23 in list. Trace execution of Search(head, 18), Insert(head, p) og Delete(head, q).

```
Node Search(Node head, int value) {
    Node x = head;
    while (x != null) {
        if (x.key == value) return x;
        x = x.next;
    }
    return null;
}

Node Insert(Node head, Node x) {
    x.prev = null;
    x.next = head;
    head.prev = x;
    return x;
}

Node Delete(Node head, Node x) {
    if (x.prev != null)
        x.prev.next = x.next;
    else head = x.next;
    if (x.next != null)
        x.next.prev = x.prev;
    return head;
}
```

Ex. Consider how to implement stack and queue with linked lists efficiently.

**Stack.** Maintain dynamic sequence (stack) S supporting the following operations:
- **Push(x):** add x to S.
- **Pop():** remove and return the most recently added element in S.
- **isEmpty():** return true if S is empty.

**Queue.** Maintain dynamic sequence (queue) Q supporting the following operations:
- **Enqueue(x):** add x to Q.
- **Dequeue():** remove and return the first added element in Q.
- **isEmpty():** return true if Q is empty.

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```

```
Stack and Queue Implementation
```
Linked Lists

- **Linked list.** Flexible data structure to maintain sequence of elements.
- Other linked data structures: Cyclic lists, trees, graphs, …

---

Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

---

Stack Implementation with Array

- **Challenge.** Can we implement a stack efficiently with arrays?
  - Do we need a fixed capacity?
  - Can we get linear space and constant time?

---

Dynamic Arrays

- **Goal.**
  - Implement a stack using arrays in \( \Theta(n) \) space for \( n \) elements.
  - As fast as possible.
  - Focus on `Push`, ignore `Pop` and `isEmpty` for now.

- **Solution 1**
  - Start with table of size 1.
  - `Push(x)`:
    - Allocate new table of size + 1.
    - Move all elements to new table.
    - Delete old table.
Dynamic Arrays

- **Push(x):**
  - Allocate new table of size + 1.
  - Move all elements to new table.
  - Delete old table.

- **Time.** Time for n Push operations?
  - ith Push takes Θ(i) tid.
  - ⇒ total time is $1 + 2 + 3 + 4 + \ldots + n = \Theta(n^2)$

- **Space.** Θ(n)

- **Challenge.** Can we do better?

---

Dynamic Arrays

- **Push(x):**
  - If table is full:
    - Allocate new table of twice the size.
    - Move all elements to new table.
    - Delete old table.

- **Time.** Time for n Push operations?
  - Push $2^k$ takes Θ($2^k$) time.
  - All other Push take Θ(1) time.
  - ⇒ total time is $1 + 2 + 4 + 8 + 16 + \ldots + 2^{\log n} + n = \Theta(n)$

- **Space.** Θ(n)

---

Dynamic Arrays

- **Idea.** Only copy elements some times

- **Solution 2.**
  - Start with table of size 1.
  - Push(x):
    - If table is full:
      - Allocate new table of twice the size.
      - Move all elements to new table.
      - Delete old table.

- **Time.** Time for n Push operations?
  - Push $2^k$ takes Θ($2^k$) time.
  - All other Push take Θ(1) time.
  - ⇒ total time is $1 + 2 + 4 + 8 + 16 + \ldots + 2^{\log n} + n = \Theta(n)$

- **Space.** Θ(n)

---

Dynamic Arrays

- **Stack with dynamic table.**
  - n Push operations in Θ(n) time and plads.
  - Extends to n Push, POP and isEmpty operations in Θ(n) time.
  - Time is amortized Θ(1) per operation.
  - With more clever tricks we can deamortize to get Θ(1) worst-case time per operation.

- **Queue with dynamic array.**
  - Similar results as stack.

- **Global rebuilding.**
  - Dynamic array is an example of global rebuilding.
  - Technique to make static data structures dynamic.
Stack and Queues

<table>
<thead>
<tr>
<th>Data structure</th>
<th>PUSH</th>
<th>POP</th>
<th>IS EMPTY</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array with capacity N</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(N))</td>
</tr>
<tr>
<td>Linked List</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>Dynamic Array 1</td>
<td>(\Theta(n)^\dagger)</td>
<td>(\Theta(1)^\dagger)</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>Dynamic Array 2</td>
<td>(\Theta(1)^\dagger)</td>
<td>(\Theta(1)^\dagger)</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>Dynamic Array 3</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>

\(\dagger = \text{amortized}\)

Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays