Introduction to Data Structures

- Data structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays
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.
  - **ISEMPTY():** return true if S is empty.

Diagram:

```
3
4
18
16
PUSH(28)
28
3
4
18
16
POP()
3
4
18
16
POP()
4
18
16
PUSH(7)
7
4
18
16
```
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.
  - **ISEMPTY():** return true if S is empty.
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 $\text{top} = -1$
- **Operations.**
  - $\text{PUSH}(x)$: Add $x$ at $S[\text{top}+1]$, $\text{top} = \text{top} + 1$
  - $\text{POP}()$: return $S[\text{top}]$, $\text{top} = \text{top} - 1$
  - $\text{ISEMPTY}()$: return true if $\text{top} = -1$.
  - Check for overflow and underflow in $\text{PUSH}$ and $\text{POP}$.

N = 10

```
16 18 4 3 28
```

\[\text{top}\]
Stack Implementation

- **Time**
  - PUSH in $\Theta(1)$ time.
  - POP in $\Theta(1)$ time.
  - ISEMPTY in $\Theta(1)$ time.

- **Space.**
  - $\Theta(N)$ space.

- **Limitations.**
  - Capacity must be known.
  - Wasting space.

---

N = 10

- Stack Implementation:

  - Array: 16, 18, 4, 3, 28, , , ,
  - Top pointer at position 5:
    - Top element is 28.

- Limitations:
  - Capacity must be known.
  - Wasting space.
Queue Implementation

- **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.

```
count = 5
N = 10

head

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>15</td>
<td>1</td>
<td>22</td>
</tr>
</tbody>
</table>
```
Queue Implementation

- **Time.**
  - ENQUEUE in $\Theta(1)$ time.
  - DEQUEUE in $\Theta(1)$ time.
  - ISEMPTY in $\Theta(1)$ time.

- **Space.**
  - $\Theta(N)$ space.

- **Limitations.**
  - Capacity must be known.
  - Wasting space.
Stacks and Queues

- **Stack.**
  - **Time.** PUSH, POP, ISEMPTY in $\Theta(1)$ time.
  - **Space.** $\Theta(N)$

- **Queue.**
  - **Time.** ENQUEUE, Dequeue, ISEMPTY in $\Theta(1)$ time.
  - **Space.** $\Theta(N)$

- **Challenge.** Can we get linear space and constant time?
Introduction to Data Structures

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

- 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

- 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.
  • 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

• Operations in Java.

```java
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. 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)`. 

```
null 7 42 18 23 5 null
```
Linked Lists

- **Time.**
  - **SEARCH** in $\Theta(n)$ time.
  - **INSERT** and **DELETE** in $\Theta(1)$ time.

- **Space.**
  - $\Theta(n)$
Stack and Queue Implementation

- 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 S is empty.
Stack and Queue Implementation

• Stacks and queues using linked lists
  • Stack.
    • **Time.** PUSH, POP, ISEMPTY in $\Theta(1)$ time.
    • **Space.** $\Theta(n)$

• Queue.
  • **Time.** ENQUEUE, Dequeue, ISEMPTY in $\Theta(1)$ time.
  • **Space.** $\Theta(n)$
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 $\Theta(i)$ tid.
  - $\Rightarrow$ total time is $1 + 2 + 3 + 4 + \ldots + n = \Theta(n^2)$

• **Space.** $\Theta(n)$

• **Challenge.** Can we do better?
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.
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 \(\Theta(2^k)\) time.
  - All other Push take \(\Theta(1)\) time.
  - \(\Rightarrow\) total time is \(1 + 2 + 4 + 8 + 16 + \ldots + 2^{\lfloor \log n \rfloor} + n = \Theta(n)\)

- **Space.** \(\Theta(n)\)
Dynamic Arrays

- Stack with dynamic table.
  - n PUSH operations in $\Theta(n)$ time and plads.
  - Extends to n PUSH, POP og ISEMPY operations in $\Theta(n)$ time.
- Time is amortized $\Theta(1)$ per operation.
- With more clever tricks we can deamortize to get $\Theta(1)$ 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>isEmpty</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array with capacity N</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Linked List</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Dynamic Array 1</td>
<td>Θ(n)†</td>
<td>Θ(1)†</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Dynamic Array 2</td>
<td>Θ(1)†</td>
<td>Θ(1)†</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Dynamic Array 3</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

† = amortized
Introduction to Data Structures

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