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.

```
3  28  4  18  16
  Push(28)  Pop()  Pop()  Push(7)
```

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.

```
8  4  15
  Enqueue(1)
  Enqueue(22)
  Dequeue()
  Dequeue()
  Enqueue(6)
```

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
    - isEmpty(): return true if top = -1.
    - Check for overflow and underflow in Push and Pop.

```
16  18  4  3  28
    top
```

N = 10
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.

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.

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.

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

- 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

- 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.

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).

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 array of size 1.
  - Push(x):
    - Allocate new array of size + 1.
    - Move all elements to new array.
    - Delete old array.
Dynamic Arrays

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

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

- **Space.** $\Theta(n)$

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

---

Dynamic Arrays

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

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

- **Space.** $\Theta(n)$

---

Dynamic Arrays

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

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

- **Space.** $\Theta(n)$

---

Dynamic Arrays

- **Stack with dynamic array.**
  - n Push operations in $\Theta(n)$ time and plads.
  - Extends to n Push, Pop and isEmpty 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>IS EMPTY</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