ETH zürich

Exercise Session 5

Data Structures and Algorithms, D-MATH, ETH Zurich

Program of today

Feedback of last exercise

Repetition theory

Programming Task

Exercise Review: "Comparing Sorting Algorithms"

Bubblesort	min	max
Comparisons	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequence	any	any
Swaps	0	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

Exercise Review: "Comparing Sorting Algorithms"

InsertionSort	min	max
Comparisons	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Swaps	0	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

SelectionSort	min	max
Comparisons	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequence	any	any
Swaps	0	$\mathcal{O}(n)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

Exercise Review: "Comparing Sorting Algorithms"

QuickSort	min	max
Comparisons	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Sequence	complex	$1, 2, \dots, n$
Swaps	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Sequence	$1, 2, \dots, n$	complex

complex: Sequence must be made such that the pivot halves the sorting range. For example ($n = 7$): $4, 5, 7, 6, 2, 1, 3$

Amortized analysis: push_back

Strategy: double if array is full.

Amortized analysis: push_back

Strategy: double if array is full.

Let $i \in \mathbb{N}$ be the number of elements appended and let $n_i \in \mathbb{N}$ be the array size allocated after appending i.

It holds that

$$n_i = \begin{cases} 1 & \text{if } i = 1 \text{ [Start]} \\ 2 \cdot n_{i-1} & \text{if } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array full]} \\ n_{i-1} & \text{otherwise} \end{cases}$$

$$n_i = 2^{\lceil \log_2 i \rceil}$$

i	n_i
1	1
2	2
3	4
4	4
5	8
6	8
..	..

Amortized analysis: push_back

Strategy: double if array is full.

[1]According to the task description: 2n initialisations, n copies, 1 new element

Amortized analysis: push_back

Strategy: double if array is full.

Real costs

$$t_i = \begin{cases} 1 & \text{if } i = 1 \text{ [Start]} \\ 3n_{i-1} + 1 & \text{if } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array full]}^1 \\ 1 & \text{otherwise} \end{cases}$$

[1]According to the task description: 2n initialisations, n copies, 1 new element

Amortized analysis: push_back

Strategy: double if array is full.

Real costs

$$t_i = \begin{cases} 1 & \text{if } i = 1 \text{ [Start]} \\ 3n_{i-1} + 1 & \text{if } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array full]}^1 \\ 1 & \text{otherwise} \end{cases}$$

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

[1] According to the task description: 2n initialisations, n copies, 1 new element

Amortized analysis: push_back

Strategy: double if array is full.

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Amortized analysis: push_back

Strategy: double if array is full.

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\begin{aligned}\Phi_i &= 6 \cdot \text{number of elements in the upper half of the array} \\ &= 6 \cdot (i - \frac{n_i}{2}) = 6i - 3n_i\end{aligned}$$

Amortized analysis: push_back

Strategy: double if array is full.

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\begin{aligned}\Phi_i &= 6 \cdot \text{number of elements in the upper half of the array} \\ &= 6 \cdot (i - \frac{n_i}{2}) = 6i - 3n_i\end{aligned}$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 6 + 3n_{i-1} - 3 \overbrace{n_i}^{2 \cdot n_{i-1}} & \text{if } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array full]} \\ 6 & \text{otherwise} \end{cases}$$

$$\Rightarrow 7 \geq a_i \text{ (in both cases)}$$

Amortized analysis: push_back

Strategy: double if array is full.

Find potential function such that the amortized costs are constant:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= \begin{cases} 3n_{i-1} + 1 + 6 - 3n_{i-1} & \text{if } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array full]} \\ 1 + 6 & \text{otherwise} \end{cases} \\ &\leq 7 \quad \text{for all } i \end{aligned}$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty.

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty.

$$t_i = \begin{cases} 1 & \text{if array is more than quarter full} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{otherwise, then } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty.

$$t_i = \begin{cases} 1 & \text{if array is more than quarter full} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4} n_{i-1} & \text{otherwise, then } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty.

$$t_i = \begin{cases} 1 & \text{if array is more than quarter full} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{otherwise, then } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Let k_i be the number of elements in the array in step i

$$\Phi_i = 3 \cdot \text{number of empty elements in the lower half of array } (1, \ldots, \frac{n}{2})$$

$$= 3 \cdot (\frac{n_i}{2} - k_i)$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty.

$$t_i = \begin{cases} 1 & \text{if array is more than quarter full} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{otherwise, then } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Let k_i be the number of elements in the array in step i

$$\Phi_i = 3 \cdot \text{number of empty elements in the lower half of array } (1, \ldots, \frac{n}{2})$$

$$= 3 \cdot (\frac{n_i}{2} - k_i)$$

$$\Rightarrow 4 \geq a_i \text{ (in both cases)}$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty. Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\Phi_i = 3 \cdot (\frac{n_i}{2} - k_i)$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 3 & \text{if array is more than quarter full} \\ 3 \cdot \left(1 + \frac{n_{i-1}}{4} - \frac{n_{i-1}}{2})\right) & \text{otherwise} \end{cases}$$

Amortized analysis: pop_back

Strategy: halve if array is three quarters empty. Find potential function such that the amortized costs are constant:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\Phi_i = 3 \cdot (\frac{n_i}{2} - k_i)$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 3 & \text{if array is more than quarter full} \\ 3 \cdot \left(1 + \frac{n_{i-1}}{4} - \frac{n_{i-1}}{2})\right) & \text{otherwise} \end{cases}$$

$$\Rightarrow 4 \geq a_i \text{ (in both cases)}$$

Amortized analysis: pop and push

$$\begin{aligned}\Phi_i = 6 &\cdot \text{number elements in the upper half} \\ &+ 3 \cdot \text{number empty slots in the lower half}\end{aligned}$$

2. Repetition theory

Hashing well-done

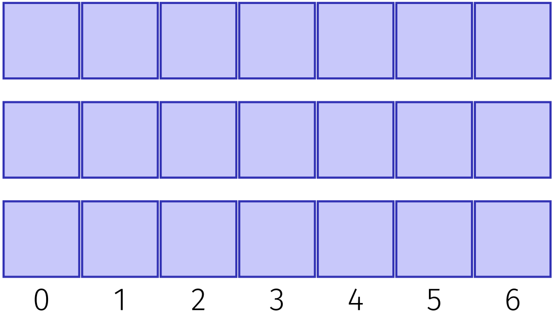
Useful Hashing...

- distributes the keys as uniformly as possible in the hash table.
- avoids probing over long areas of used entries (e.g. primary clustering).
- avoids using the same probing sequence for keys with the same hash value (e.g. secondary clustering).

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

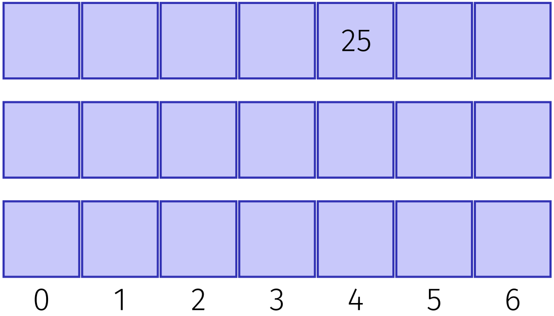
- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

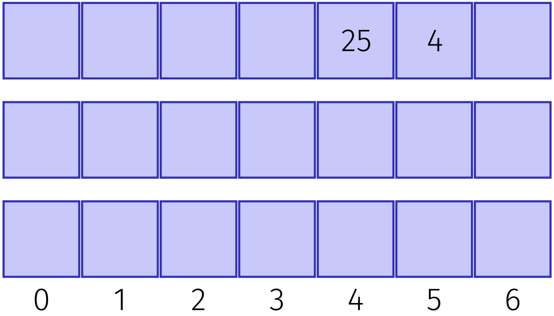
- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

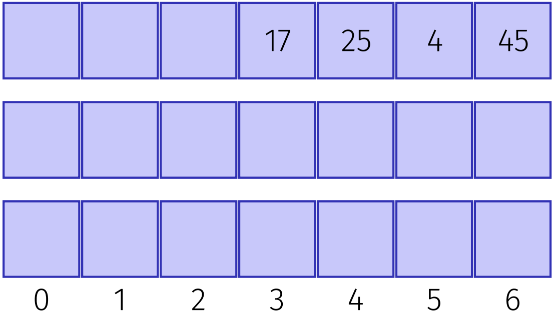
- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.

0	1	2	3	4	5	6
			17	25	4	

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing, $s(j, k) = j$.
- quadratic probing, $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing, $s(j, k) = j \cdot (1 + (k \bmod 5))$.

0	1	2	3	4	5	6
			17	25	4	45
				25		

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing, $s(j, k) = j$.
- quadratic probing, $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing, $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
				25	4	
0	1	2	3	4	5	6

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing, $s(j, k) = j$.
- quadratic probing, $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing, $s(j, k) = j \cdot (1 + (k \bmod 5))$.

0	1	2	3	4	5	6
			17	25	4	45

0	1	2	3	4	5	6
			17	25	4	

0	1	2	3	4	5	6

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j,k)$:

- linear probing,
  $s(j,k) = j$.
- quadratic probing,
  $s(j,k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j,k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
0	1	2	3	4	5	6

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.

0	1	2	3	4	5	6
			17	25	4	45
		45	17	25	4	
				25		

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.

	0	1	2	3	4	5	6
linear probing				17	25	4	45
quadratic probing			45	17	25	4	
Double Hashing			4		25		

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j,k)$:

- linear probing,
  $s(j,k) = j$.
- quadratic probing,
  $s(j,k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j,k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
		4	17	25		
0	1	2	3	4	5	6

Hashing Examples

Insert the keys $25, 4, 17, 45$ into the hash table, using the function $h(k) = k \bmod 7$ and probing to the right, $h(k) + s(j, k)$:

- linear probing,
  $s(j, k) = j$.
- quadratic probing,
  $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
  $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
		4	17	25	45	
0	1	2	3	4	5	6

Simple Uniform Hashing

Statement about the uniform distribution and independence of the **keys**.

Property of closed addressing: simple uniform hashing $\Rightarrow$ expected length of the chains as good as possible $\leq \alpha = \frac{n}{m}$.

Uniform Hashing

Statement about the uniform distribution and independence of **key probing sequences**.

Property of open addressing: Uniform Hashing $\Rightarrow$ expected runtime costs $\leq \frac{1}{1-\alpha}$.

Universal Hashing

Property about the available, randomly chosen **hash-functions**

$$|\{h \in \mathcal{H} \text{ with } h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}$$

Property independent of chose sequence of keys: for hashing with chaining the expected chain length is $\leq \alpha = \frac{n}{m}$

Prerequisite for Perfect Hashing

3. Programming Task

Finding a Sub-Array

- Given: two integer arrays $A = (a_0, \ldots, a_{n-1})$ and $B = (b_0, \ldots, b_{k-1})$
- Task: Find position of B in A.

Finding a Sub-Array

- Given: two integer arrays $A = (a_0, \ldots, a_{n-1})$ and $B = (b_0, \ldots, b_{k-1})$
- Task: Find position of B in A.
- Naive: Loop through A, check whether the following k entries match B.

Finding a Sub-Array

- Given: two integer arrays $A = (a_0, \ldots, a_{n-1})$ and $B = (b_0, \ldots, b_{k-1})$
- Task: Find position of B in A.
- Naive: Loop through A, check whether the following k entries match B.
  - $O(nk)$ comparison operations

Finding a Sub-Array

- Given: two integer arrays $A = (a_0, \ldots, a_{n-1})$ and $B = (b_0, \ldots, b_{k-1})$
- Task: Find position of B in A.
- Naive: Loop through A, check whether the following k entries match B.
  - $O(nk)$ comparison operations
- Solution using hashing: Calculate hash $h(B)$ and compare it to $h((a_i, a_{i+1}, \ldots, a_{i+k-1}))$.
- Avoid re-computing $h((a_i, a_{i+1}, \ldots, ai + k - 1)$ for each $i \implies O(n)$ expected

Sliding Window Hash

- Possible hash function: sum of all elements:
  - Can be updated easily: subtract a_i and add a_{i+k}.
  - However: bad hash function

Sliding Window Hash

- Possible hash function: sum of all elements:
  - Can be updated easily: subtract a_i and add a_{i+k}.
  - However: bad hash function
- Better:

$$H_{c,m}((a_i, \cdots, a_{i+k-1})) = \left(\sum_{j=0}^{k-1} a_{i+j} \cdot c^{k-j-1} \right) \bmod m$$

  - $c = 1021$ prime number
  - $m = 2^{15}$ **`int`**, no overflows at calculations

Computing with Modulo

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$
$$(a - b) \bmod m = ((a \bmod m) - (b \bmod m) + m) \bmod m$$
$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

Exercise: Compute

$$12746357 \bmod 11$$

Computing Modulo

Exercise: Compute

$$12746357 \bmod 11$$

Computing Modulo

Exercise: Compute

$$
\begin{aligned}
&12746357 \bmod 11 \\
&= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11
\end{aligned}
$$

Computing Modulo

Exercise: Compute

$$
\begin{aligned}
&12746357 \bmod 11 \\
&= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11 \\
&= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11
\end{aligned}
$$

For the second equality we used the fact that $10^2 \bmod 11 = 1$.

Computing Modulo

Exercise: Compute

$$
\begin{aligned}
&12746357 \bmod 11 \\
&= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11 \\
&= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11 \\
&= (7 + 6 + 3 + 5 + 4 + 4 + 2 + 10) \bmod 11
\end{aligned}
$$

For the second equality we used the fact that $10^2 \bmod 11 = 1$.

Computing Modulo

Exercise: Compute

$$
\begin{aligned}
&12746357 \bmod 11 \\
&= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11 \\
&= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11 \\
&= (7 + 6 + 3 + 5 + 4 + 4 + 2 + 10) \bmod 11 \\
&= 8 \bmod 11.
\end{aligned}
$$

For the second equality we used the fact that $10^2 \bmod 11 = 1$.

Sliding Window Hash

```
template<typename It1, typename It2>
It1 findOccurrence(const It1 from, const It1 to,
                   const It2 begin, const It2 end)
{
  const unsigned k = end - begin;
  const unsigned M = 32768;
  const unsigned C = 1021;

  // your code here
  // ...
```

Sliding Window Hash

```
  // elements can be compared using std::equal:
  if(std::equal(window_left, window_right, begin, end))
    return current;

  // if no occurrence is found return end of array
  return to;
}
```

Sliding Window Hash

Make sure that

- the algorithm computes c^k only once,
- all computations are modulo m for all values in order not to get an overflow (recall the rules of modular arithmetic), and
- the values are always positive (e.g., by adding multiples of m).

Questions?