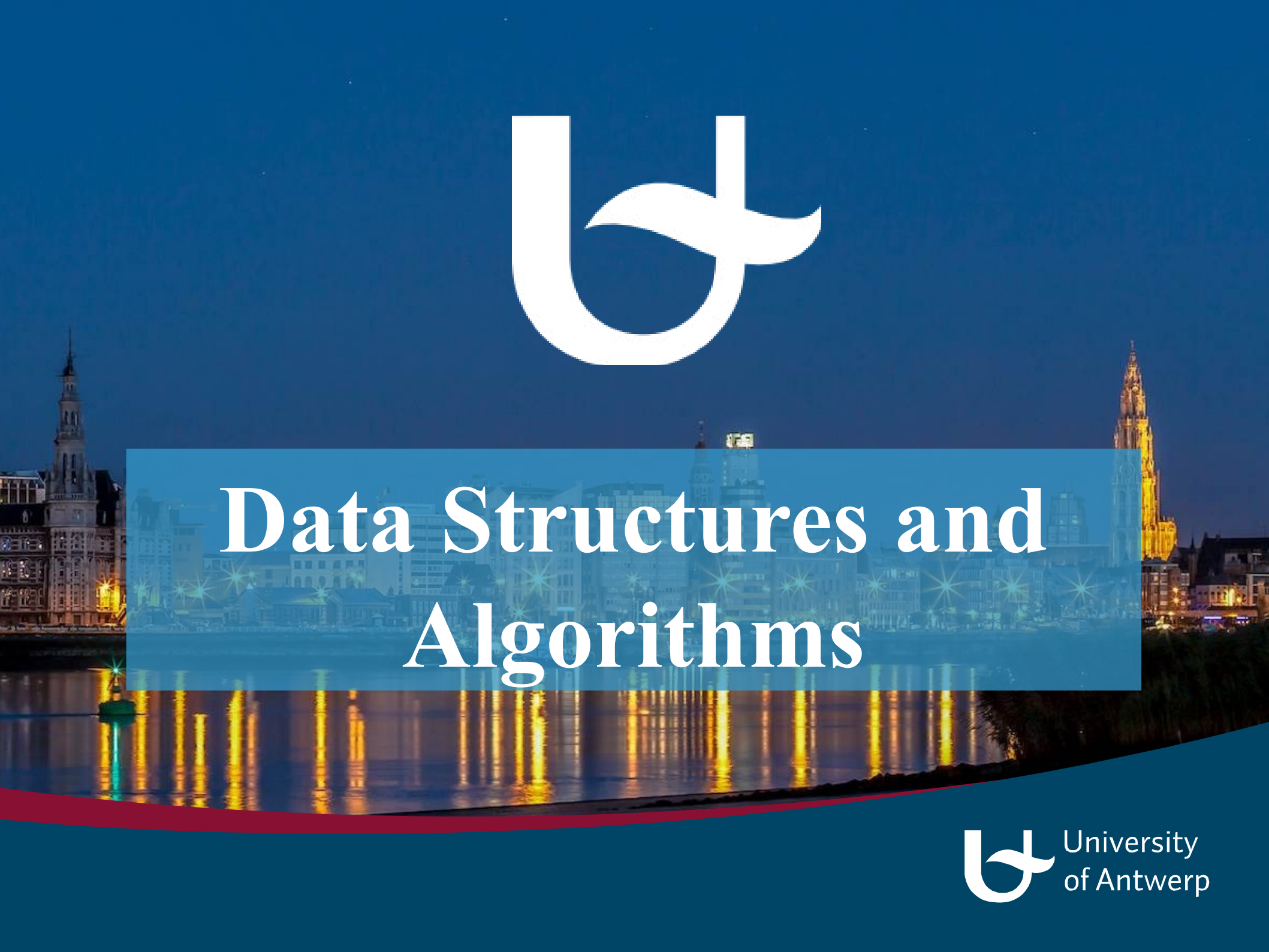


Data Structures and Algorithms

University of Antwerp

Maps and Hashing

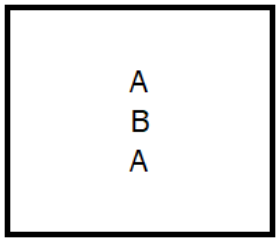
Lecturer: Dr. Ali Anwar

Objectives

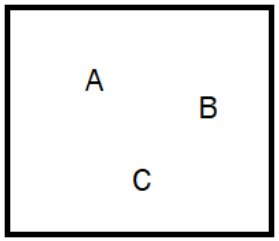
- Introduction of
  - Maps
  - Hash Functions
  - Hash Tables
  - Dictionaries
  - Priority Queues

Data Structures

- **Lists** - have ordering for their elements
  - Arrays
  - Stacks
  - Linked Lists
  - Queues

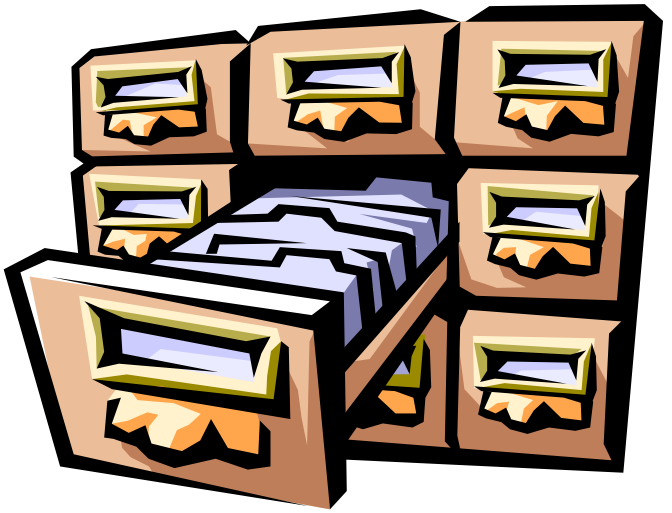


- **Sets** - don't have ordering, but instead don't allow for repeated elements
  - Maps
  - Dictionaries
  - Priority Queues



Part 1

Maps



Maps

- A map stores a collection of *(key,value)* pairs, such that each possible key appears at most once in the collection.
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

Entry ADT

- An entry stores a key-value pair (k, v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v

The Map ADT

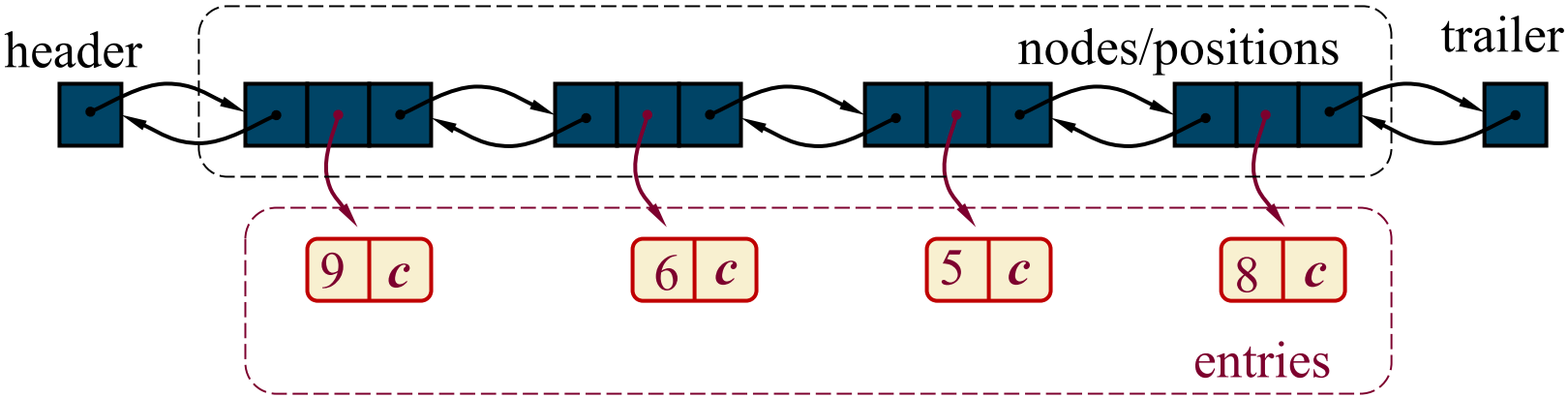
- find(*k*): if the map *M* has an entry with key *k*, return and iterator to it; else, return special iterator end
- put(*k, v*): if there is no entry with key *k*, insert entry (*k, v*), and otherwise set its value to *v*. Return an iterator to the new/modified entry
- erase(*k*): if the map *M* has an entry with key *k*, remove it from *M*
- size(), empty()
- begin(), end(): return iterators to beginning and end of *M*

Example

Operation	*Output*	*Map*
empty()	**true**	Ø
put(5,A)	[(5,A)]	(5,A)
put(7,B)	[(7,B)]	(5,A),(7,B)
put(2,C)	[(2,C)]	(5,A),(7,B),(2,C)
put(8,D)	[(8,D)]	(5,A),(7,B),(2,C),(8,D)
put(2,E)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
find(7)	[(7,B)]	(5,A),(7,B),(2,E),(8,D)
find(4)	end	(5,A),(7,B),(2,E),(8,D)
find(2)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
erase(5)	—	(7,B),(2,E),(8,D)
erase(2)	—	(7,B),(8,D)
find(2)	end	(7,B),(8,D)
empty()	**false**	(7,B),(8,D)

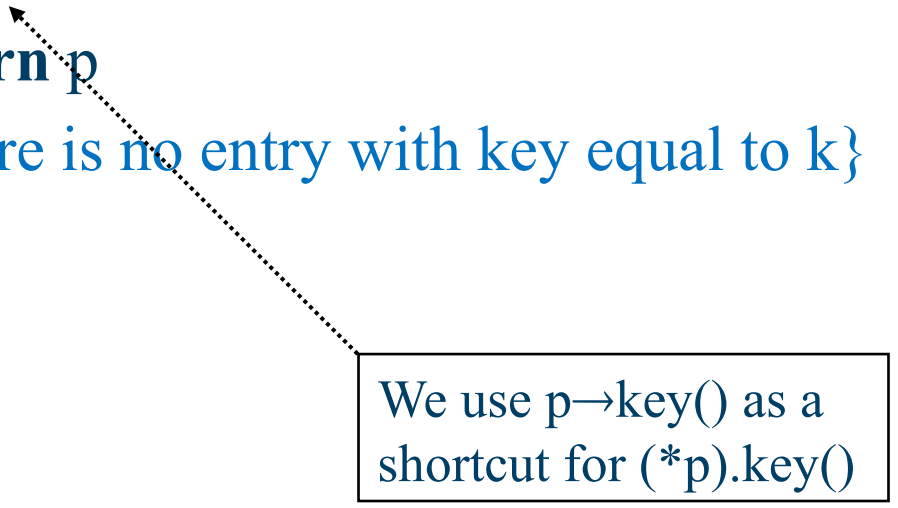
A Simple List-based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The *find* Algorithm

```
Algorithm find(k):
        for each p in [S.begin(), S.end()) do
                if p→key() = k then
                        return p
        return S.end() {there is no entry with key equal to k}
```



The *put* Algorithm

```
Algorithm put(k,v):
        for each p in [S.begin(), S.end()) do
                if p→key() = k then
                        p→setValue(v)
                        return p
        p = S.insertBack((k,v)) {there is no entry with key k}
        n = n + 1   {increment number of entries}
        return p
```

The *erase* Algorithm

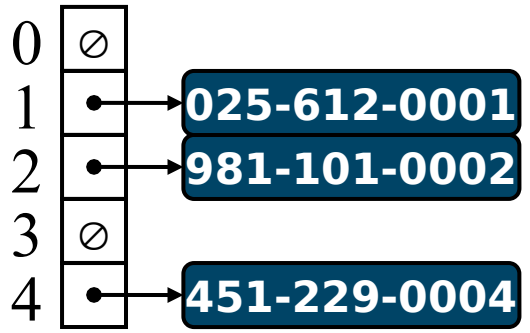
```
Algorithm erase(k):
    for each p in [S.begin(), S.end()) do
        if p.key() = k  then
            S.erase(p)
            n = n – 1          {decrement number of entries}
```

Performance of a List-Based Map

- Performance:
  - *put* takes O(n) time since we need to determine whether it is already in the sequence
  - *find* and *erase* take O(n) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Part 2

Hashing



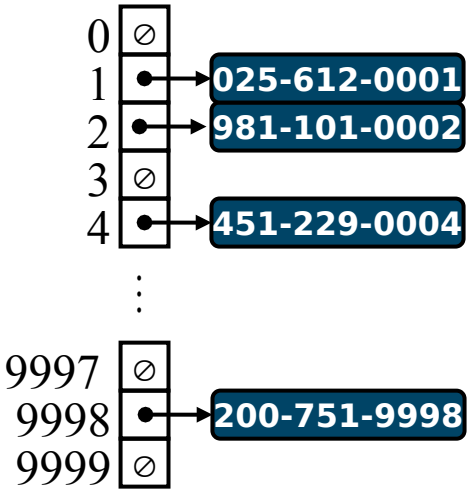
Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example: $h(x) = x \bmod N$ is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x)$ = last four digits of x



Hash Functions

- A hash function is usually specified as the composition of two functions:
- Hash code:
  h_1 : keys → integers
- Compression function:
  h_2 : integers → [0, N - 1]

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes

- Memory address:
  - We reinterpret the memory address of the key object as an integer
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in C++)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in C++)

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $a_0 a_1 \ldots a_{n-1}$

  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

    at a fixed value $z != 1$, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Polynomial $p(z)$ can be evaluated in O(n) time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in O(1) time

    $p_0(z) = a_{n-1}$

    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$

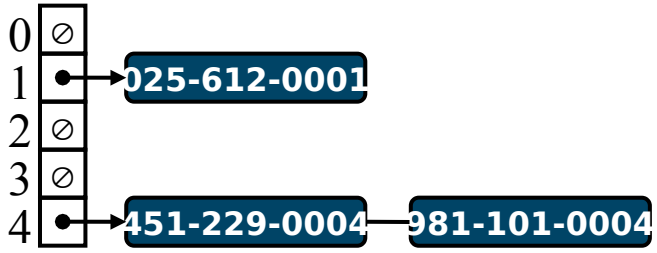
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Map with Separate Chaining

Delegate operations to a list-based map at each cell:

Algorithm find(k):
return A[h(k)].find(k)

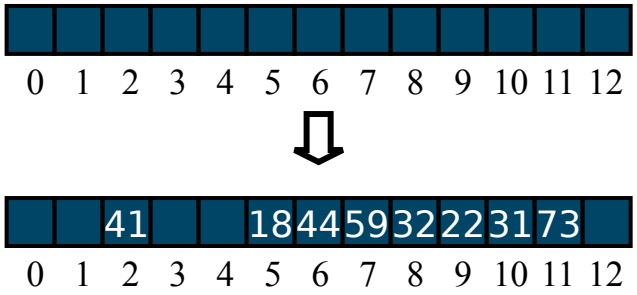
Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = **null then** {k is a new key}
    A[h(k)].insert(k,v) // add a link to the linked list at h(k)
    n = n + 1
return t

Algorithm erase(k):
t = A[h(k)].erase(k)
if t ≠ **null then** {k was found}
    n = n - 1
return t

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing

- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key() = k
            return c.value()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements
- erase(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item *AVAILABLE* and we return element o
  - Else, we return *null*
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

  $$(i + jd(k)) \bmod N$$

  where $i = h(k)$

  for $j = 0, 1, \ldots, N - 1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:

  $$d_2(k) = q - k \bmod q$$

  where

  - $q < N$
  - q is a prime

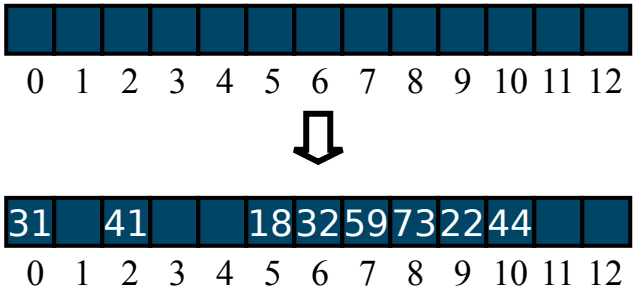
- The possible values for $d_2(k)$ are

  $$1, 2, \ldots, q$$

Example of Double Hashing (H.W)

- Consider a hash table storing integer keys that handles collision with double hashing
  - *N = 13*
  - *h(k) = k mod 13*
  - *d(k) = 7 - k mod 7*
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

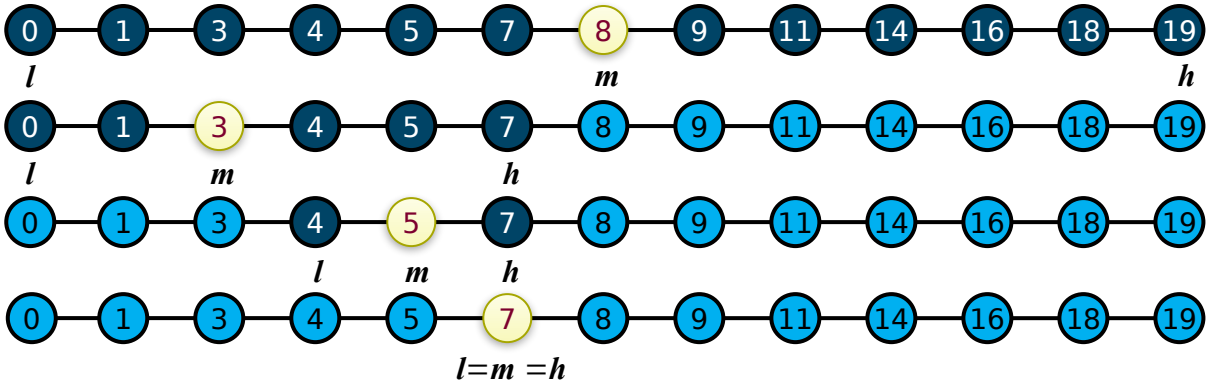
- In the worst case, searches, insertions and removals on a hash table take O(n) time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $a = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$1 / (1 - a)$$

- The expected running time of all the dictionary ADT operations in a hash table is O(1)
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Part 3

Dictionaries



Dictionary

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

EntryADT

- An entry stores a key-value pair (k,v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v

Dictionary ADT

- Dictionary ADT methods:
  - find(*k*): if there is an entry with key *k*, returns an iterator to it, else returns the special iterator end
  - findAll(*k*): returns iterators *b* and *e* such that all entries with key *k* are in the iterator range [*b, e*) starting at *b* and ending just prior to *e*
  - put(*k, o*): inserts and returns an iterator to it
  - erase(*k*): remove an entry with key *k*
  - begin(), end(): return iterators to the beginning and end of the dictionary
  - size(), empty()

Example

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(4)	**end**	(5,A),(7,B),(2,C),(8,D),(2,E)
find(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
findAll(2)	{(2,C),(2,E)}	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
erase(5)	—	(7,B),(2,C),(8,D),(2,E)
find(5)	**end**	(7,B),(2,C),(8,D),(2,E)

A List-based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - *put* takes O(1) time since we can insert the new item at the beginning or at the end of the sequence
  - *find* and *erase* take O(n) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

The *find*, *put*, *erase* Algorithms

```
Algorithm find(k)
for each p in [S.begin(), S.end()) do
        if p.key() = k  then
                return p

Algorithm put(k, v)
Create a new entry e = (k, v)
p = S.insertBack(e)     {S is unordered}
return p

Algorithm erase(k):
for each p in [S.begin(), S.end()) do
        if p.key() = k then
                S.erase(p)
```

Hash Table Implementation

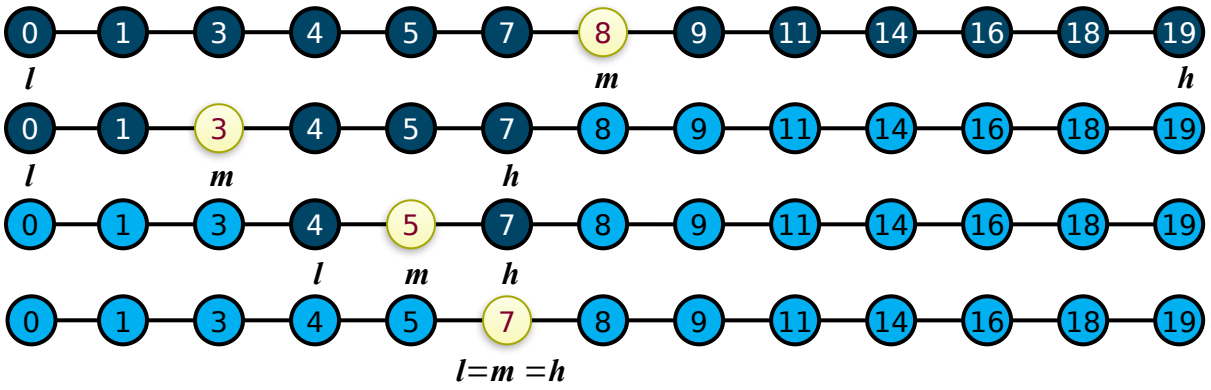
- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Search Table

- A search table is a dictionary implemented by means of a sorted array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - *find* takes O(log n) time, using binary search
  - *put* takes O(n) time since in the worst case we have to shift n/2 items to make room for the new item
  - *erase* takes O(n) time since in the worst case we have to shift n/2 items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search

- Binary search performs operation *find*(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example: *find*(7)



Part 4

Priority Queues

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Supports arbitrary element insertion but supports removal in order of priority
- Main methods of the Priority Queue ADT
  - *insert*(*e*) inserts an entry *e*
  - *removeMin*() removes the entry with smallest key
- Additional methods
  - *min() returns (but does not remove) an entry with smallest key*
  - *size(), empty()*
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - *Reflexive property:*
    $x \leq x$
  - *Antisymmetric property:*
    $x \leq y \wedge y \leq x \Rightarrow x = y$
  - *Transitive property:*
    $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Priority Queue - Example

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

Comparator ADT

- Implements the boolean function *isLess*(*p*,*q*), which tests whether $p < q$
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!*isLess*(p, q) && !*isLess*(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
      bool operator()(const Point2D& p,
      const Point2D& q) const
      { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
      bool operator()(const Point2D& p,
      const Point2D& q) const
      { return p.getY() < q.getY(); }
};
```

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of *removeMin* operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with
comparator *C*
while ¬*S.empty* ()
e ← *S.front*();
S.eraseFront()
P.insert (*e*, ⊘)
while ¬*P.empty*()
e ← *P.removeMin*()
S.insertBack(*e*)

Sequence-based Priority Queue

- Implementation with an **unsorted list**

- Performance:
  - *insert* takes O(1) time since we can insert the item at the beginning or end of the sequence
  - *removeMin* and *min* take O(n) time since we have to traverse the entire sequence to find the smallest key

- Implementation with a **sorted list**

- Performance:
  - *insert* takes O(n) time since we have to find the place where to insert the item
  - *removeMin* and *min* take O(1) time, since the smallest key is at the beginning

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n *insert* operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n *removeMin* operations takes time proportional to
     $$1 + 2 + \ldots + n$$
- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Select min at each step

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n *insert* operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n *removeMin* operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

		Sequence S	Priority queue P
Input:		(7,4,8,2,5,3,9)	()
Phase 1			
	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(4,7)
	(c)	(2,5,3,9)	(4,7,8)
	(d)	(5,3,9)	(2,4,7,8)
	(e)	(3,9)	(2,4,5,7,8)
	(f)	(9)	(2,3,4,5,7,8)
	(g)	()	(2,3,4,5,7,8,9)
Phase 2			
	(a)	(2)	(3,4,5,7,8,9)
	(b)	(2,3)	(4,5,7,8,9)
	..	..	..
	(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence