ALGORITHMS – COMTEK3,CCT3 & ESD3
Simple Data Structures & Hash Tables

Ramoni Adeogun
Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

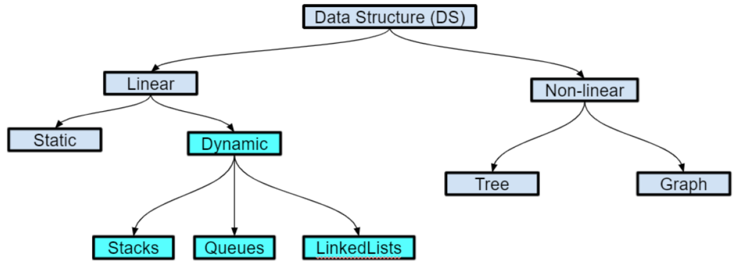
AALBORG UNIVERSITET

Outline

Simple Data Structures

Introduction

- A data structure is a
  - way of organizing and storing data in a computer so that it can be accessed and used efficiently.
  - representation of static and/or dynamic sets



- Why do we need to study data structures? Organization and Efficiency
  - DS contributes significantly to algorithm efficiency.
  - We can store the data in arrays then why do we need linked lists and other data structures?

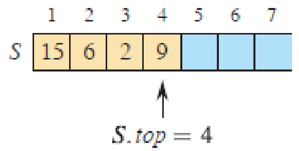
    Several operations such as add, delete, update, and search take longer to perform on arrays than other data types

Stacks

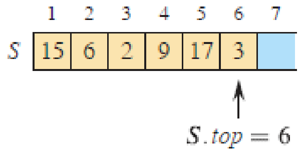
- Stacks are dynamic sets implementing a pile of elements
- Stacks implement a **last-in, first out (LIFO)** policy for insertion and deletion of elements
    - Insertion of an element occurs at the top of the stack
    - Deletion of an element also occur at the top of the stack
- Stacks are also called **pushdown list**.
- Interface
    - Stack-Empty(S) returns true if and only if S is empty
    - Push(S, x) pushes the value x onto the stack S
    - Pop(S) extracts and returns the value on the top of the stack S
- Implementation
    - using an array or list
    - using a linked list

Stack Implementation

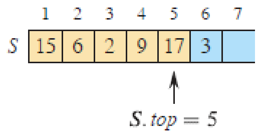
- Array (List)-based implementation
  - S is an array that holds the elements of the stack
  - top(S) is the current position of the top element of S
- Note
  - Performing the PUSH operation on a stack with $S.top = S.length$, the stack overflow!!
  - Pop an empty stack result, the stack underflow!!
- Example:



PUSH(S,17)



PUSH(S,3)



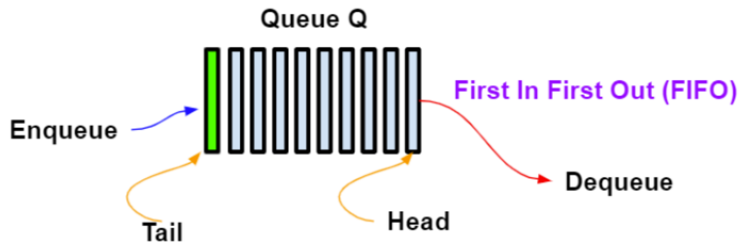
```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

PUSH(S, x)
1  if S.top == S.size
2      error "overflow"
3  else S.top = S.top + 1
4      S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top − 1
4      return S[S.top + 1]
```

AALBORG UNIVERSITET

Queues

- A queue is a dynamic set in which all items are:
  - inserted at one end (rear or the back or the tail) and
  - deleted at the other end (front or the head)
  - follows **First In First Out (FIFO)** principle

- Examples:
  - a Printer Queue,
  - a Queue of customers in a bank



- Interface
  - Enqueue(Q, x) adds element x at the back of queue Q
  - Dequeue(Q) extracts the element at the head of queue Q

Array Implementation of Queues

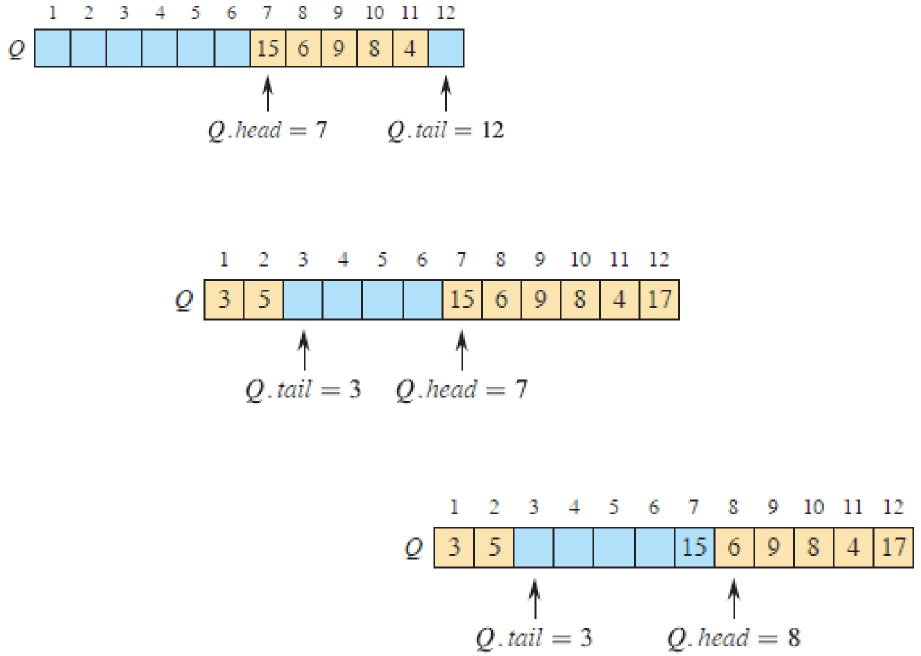
- We can implement a queue with array $Q[1, \cdots, n]$
  - Q holds at most n elements
  - enqueueing more than Q elements causes an overflow error
  - dequeueing an empty queue causes an underflow error

- The array has two attributes:
  - Q.head indexing the oldest element
  - Q.tail which points to the next available element

- The elements in a queue are at locations: $Q.head, Q.head + 1, \cdots, Q.tail - 1$
- The indices are wrapped around with the index 1 immediately following index n in a circular manner

Array Implementation of Queues

- Example:



```
ENQUEUE($Q, x$)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.size$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$

DEQUEUE($Q$)
1  $x = Q[Q.head]$
2  if $Q.head == Q.size$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

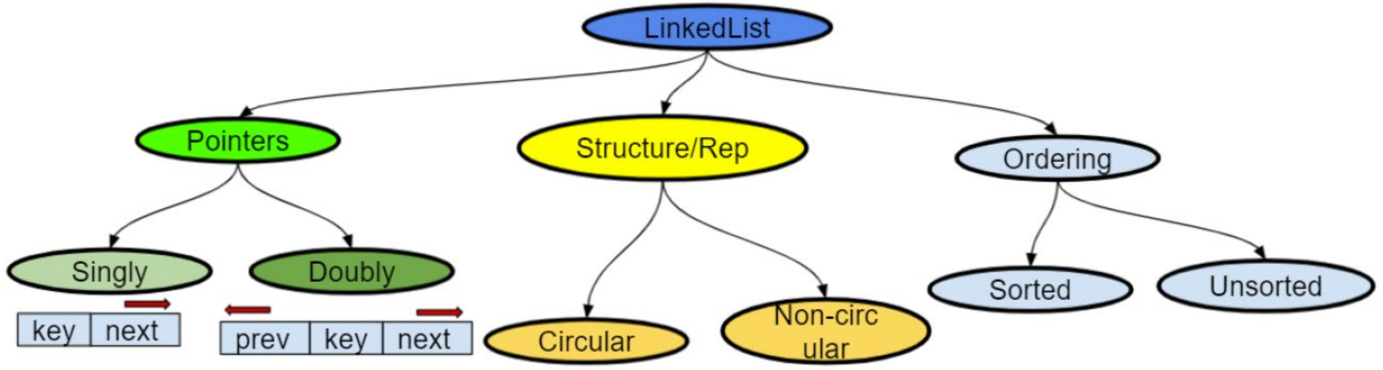
Linked Lists

- A list (or sequence) is a container that stores elements in a linear order and can involve virtually anything, e.g.,
  - a shopping list [bread, butter, cookies, ananas]
  - a list of integers [6, 8, 9, 3, 2, 5]
- Why use linked lists?
  - Allows to store data structures in multiple, non-sequential memory blocks => memory efficiency
  - Simple insert and deletion of links ! time efficiency
- Lists can be represented on different levels:
  - an abstract level allowing us to define what we mean
  - a level on which computers can communicate,
  - a machine level for implementation
- Non-empty lists can be represented by two-cells:

  1. pointer to a list element,
  2. pointer to either the empty list or another two-cell

Linked List: Classifications

- Classification of linked lists

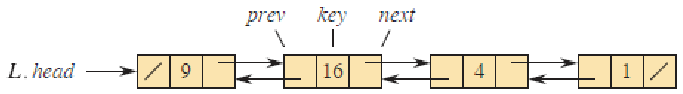


- Circular list → **ring of elements**
  - prev(L.head) → tail
  - next(L.tail) → head
- If a list is **sorted**
  - linear order of the list corresponds to linear order of keys stored in elements of the list

Linked List Interfaces

- Searching a linked list

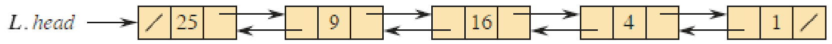
```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

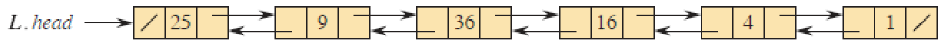


- Insert into a linked list

```
LIST-PREPEND(L, x)
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

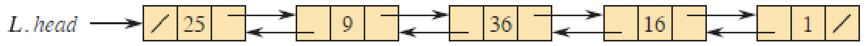
```
LIST-INSERT(x, y)
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```





- Deleting from a linked list

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```



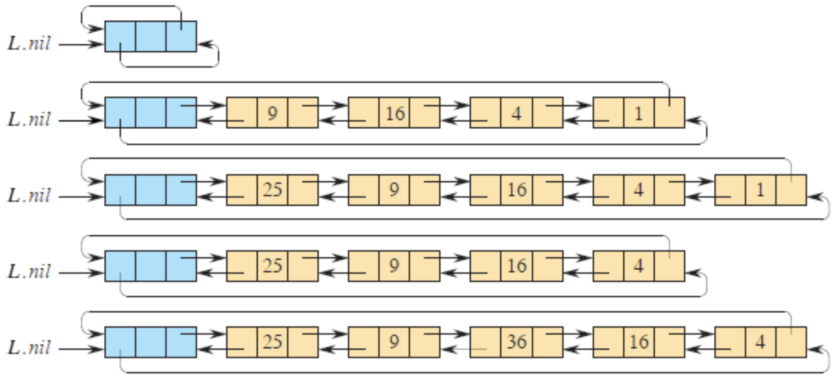
Sentinels

- A sentinel is a dummy object for simplifying boundary conditions at the headand tail of a list.
- Simplified linked list interfaces:

```
LIST-DELETE'(x)
1  $x.prev.next = x.next$
2  $x.next.prev = x.prev$
```

```
LIST-INSERT'(x, y)
1  $x.next = y.next$
2  $x.prev = y$
3  $y.next.prev = x$
4  $y.next = x$
```

```
LIST-SEARCH'(L, k)
1  $L.nil.key = k$
2  $x = L.nil.next$
3  while $x.key \neq k$
4      $x = x.next$
5  if $x == L.nil$
6      return NIL
7  else return $x$
```



Representing rooted trees

- Linked lists work well for representing linear relationships, **but not all relationships are linear**.
- How then do we represent non-linear relationships? rooted trees represented by linked data structures.

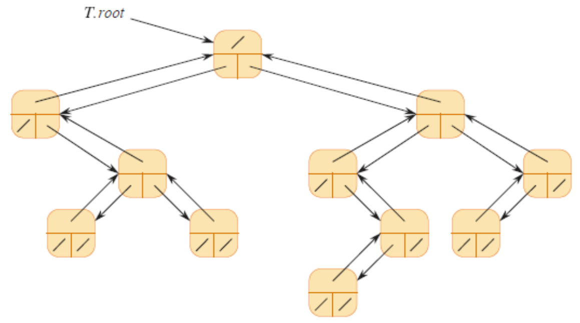
- We can use:
  - **Binary trees**,
  - **Rooted trees with unbounded branching**: rooted trees in which nodes can have an arbitrary number of children.
  - **Heaps**
- We represent each node of a tree by an object.
  - As with linked lists, we assume that **each node contains a key attribute**.
  - The remaining attributes of interest are **pointers to other nodes**, and they vary according to the type of tree.

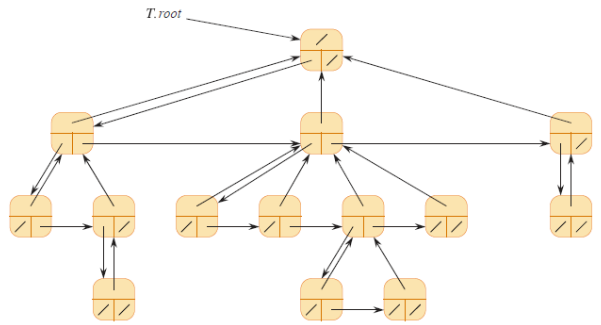
Rooted Trees

- for each node $x \neq root(T)$, parent(x) is x's parent node
- Binary trees:
  - Fixed branching

    left-child(x) and right-child(x)
- Rooted tree with unbounded branching
  - We can have all imaginable topologies of links,

    e.g., pointers to parents, left sibling and right sibling
  - left-child(x) is x's first (leftmost) child
  - right-sibling(x) is x closest sibling to the right





AALBORG UNIVERSITET

Hash Tables

Motivation

- Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.
  - Example: a symbol table in a compiler.
- A hash table is effective for implementing a dictionary.
  - Expected time to search for an element is $O(1)$
  - Worst-case search time is $\Theta(n)$.
- A hash table is a generalization of an ordinary array.
  - With an ordinary array, we store the element whose key is k in position k of the array.
  - Given a key k, we find the element whose key is k by just looking in the kth position of the array.

    This is called **direct addressing**.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

Hash Tables- Introduction

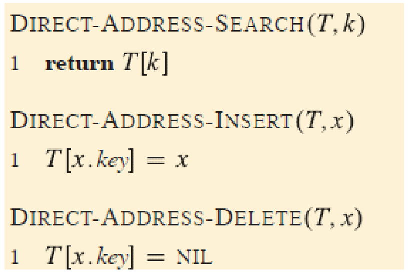
- We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.
  - Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
  - A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
  - Given a key k, don't just use k as the index into the array.

    Instead, compute a function of k, and use that value to index into the array. We call this function a **hash function**.

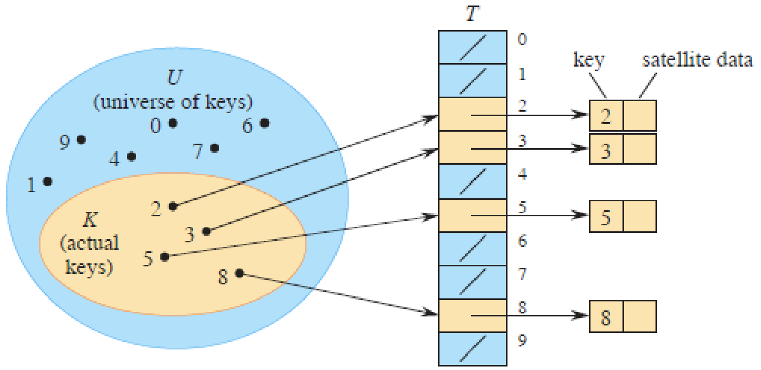
- Issues to be addressed:
  - How to compute hash functions?
  - What to do when the hash functions maps multiple keys to the same table entry?

Direct-address table

- Scenario
  - Each element has a key drawn from a universe $U = \{0,1,\dots,m\}$ where m isn't too large and no two elements have the same key.



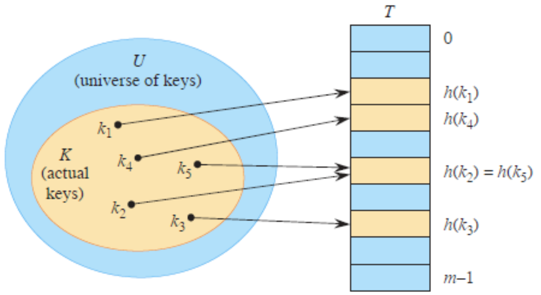
```
DIRECT-ADDRESS-SEARCH($T,k$)
1  return $T[k]$

DIRECT-ADDRESS-INSERT($T,x$)
1  $T[x.key] = x$

DIRECT-ADDRESS-DELETE($T,x$)
1  $T[x.key] =$ NIL
```

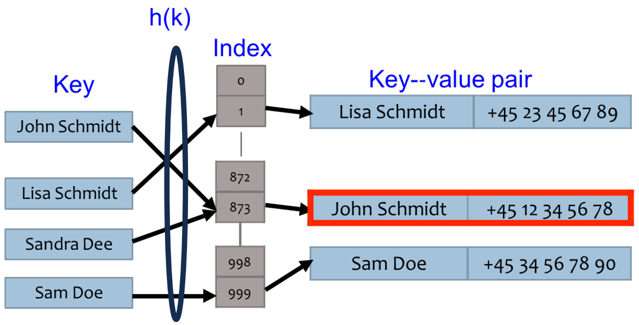


- Represent by a direct-address table, or array, $T[0,1,\dots,m]$
  - Each slot, or position, corresponds to a key in U.
  - If there's an element x with key k, then $T[k]$ contains a pointer to x.
    Otherwise, $T[k]$ is empty, represented by NIL.

Hash Tables

- The core idea of hash tables is to use a function to map keys to positions in the table T (e.g., an array).
  - An element with key k, ”hashes” to slot h(k)
$$h: U \rightarrow \{0,1,\dots,m\}$$
  - The size of the array T can be reduced, compared to the direct addressing.
- Average search time is $O(1)$.
- A challenge is that keys can collide (i.e., hash to same position in array.)
  - **Solution**: Chaining pr open addressing!



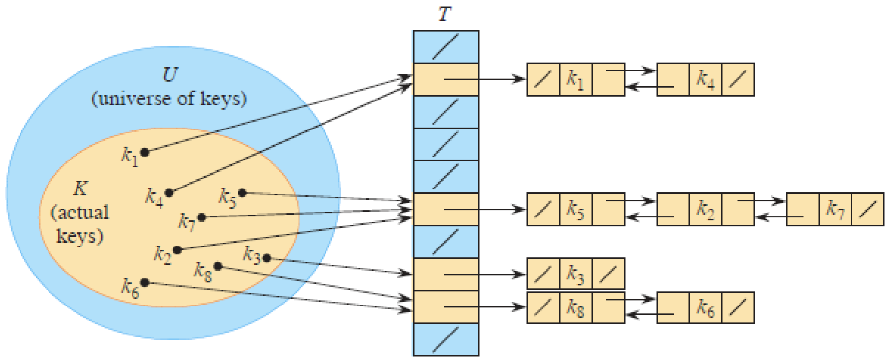


How to avoid collisions? -> How do we ensure that Sandra Dee is being contained as well as John Schmidt?

Collision resolution by chaining

- Put all elements that hash to the same slot into a linked list.
- Implementation of dictionary operations

```
CHAINED-HASH-INSERT$(T, x)$
1  LIST-PREPEND$(T[h(x.key)], x)$

CHAINED-HASH-SEARCH$(T, k)$
1  return LIST-SEARCH$(T[h(k)], k)$

CHAINED-HASH-DELETE$(T, x)$
1  LIST-DELETE$(T[h(x.key)], x)$
```



AALBORG UNIVERSITET

Analysis of hashing with chaining

- Given a key, **how long does it take to find an element with that key**, or to determine that **there is no element with that key**?

- Analysis is in terms of the **load factor** $\alpha = n/m$:
  - n is the number of elements in the table.
  - m is the number of slots in the table = number of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.

    Can have $\alpha < 1, \alpha = 1$ or $\alpha > 1$.

- Worst case is when all n keys hash to the same slot → get a single list of length n → worst-case time to search is $\Theta(n)$ plus time to compute hash function.

- Average case depends on how well the hash function distributes the keys among the slots.

Analysis of hashing with chaining

We focus on average-case performance of hashing with chaining.

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots
- For $j = 0,1, \dots, m-1$, denote the length of list $T[j]$ by n_j . Then $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

Theorem: A successful/an unsucessful search takes expected time $\Theta(1 + \alpha)$

Proof is left as an exercise.

Hash functions

- **What makes a good hash function?**
  - Ideally, the hash function satisfies the assumption of **simple uniform hashing**
  - In practice, it's not possible to satisfy this assumption, since we don't know in advance the **probability distribution that keys are drawn from**, and the keys may not be drawn independently.
  - Often use heuristics, based on the domain of the keys, to create a hash function that performs well.
- **Keys as natural numbers**
- Hash functions assume that the keys are natural numbers.
  - When they're not, have to interpret them as natural numbers.
- Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: $C = 67, L = 76, R = 82, S = 83$
  - There are 128 basic ASCII values.
    - we interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141764947$

Division method

$$h(k) = k \bmod m$$

- **Example**: $m = 20$ and $k = 91 \rightarrow h(k) = 1$
- **Advantage**:
  - Fast computation => requires only a single division
- **Disadvantage**: Have to avoid certain values of m:
  - Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
  - If k is a character string interpreted in radix 2^p (as in CLRS example), then $m = 2^p - 1$ is bad

- **Good choice for m**: A prime not too close to an exact power of 2

Multiplication method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

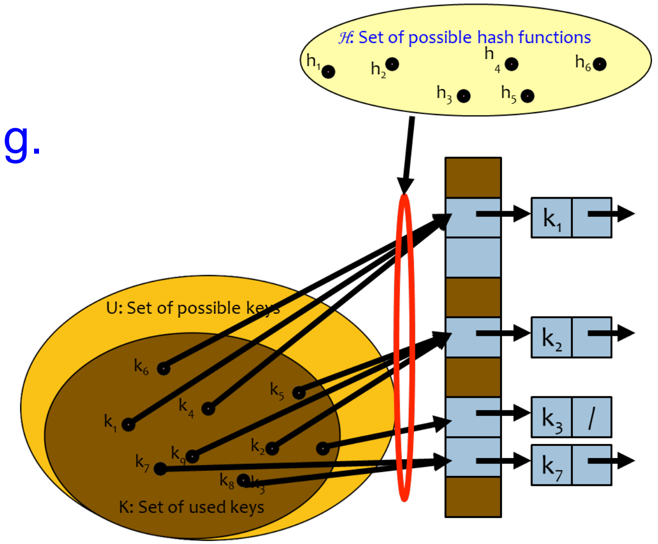
- where: $kA \bmod 1 = kA - \lfloor kA \rfloor$ = fractional part of kA
- Advantage: Value of m is not critical
- Disdvantage: Sloer than division method

- How to choose A:
    - The multiplication method works with any legal value of A.
    - But it works better with some values than with others, depending on the keys being hashed.
    - Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

- Example:

Universal hashing

- A potential problem is that a malicious adversary can deliberately choose n keys that result in worst case search time Θ(n). Possibly leading to denial of service.
- With universal hashing, a random hash function from the set of hash functions H is used every time.
- This allows us to avoid worst case situations.
- Collisions are (as before) avoided with chaining.



AALBORG UNIVERSITET

Designing universal hash functions

- Select a prime *p* large enough that any possible *k* is found in the range 0 to p-1
- Let $\mathbf{Z}_p$ be the set {0, 1, …., *p - 1*} and $\mathbf{Z}_p^*$ be the set {1, 2, … , *p - 1*}
- We further assume that
  - The amount of possible keys are larger than the amount of slots, p > m
- We define a hash function $h_{a,b}$, with a ∈ $\mathbf{Z}_p$ and b ∈ $\mathbf{Z}_p^*$
  - $h_{a,b}(k) = ((ak + b)\ mod\ p)\ mod\ m$
  - E.g. with p = 17 and m =6, we will have $h_{3,4}(8) = 5$
- The family of such functions is denoted as:
  - $H_{p,m}$ = {$h_{a,b}$: a ∈ $\mathbf{Z}_p$ og b ∈ $\mathbf{Z}_p^*$} and can be shown to be universal
- Number of slots (m) **needs not to be a prime**!

- <u>**The resulting hash is random and NOT dependent on stored keys!!**</u>

Open addressing

- An alternative to chaining for handling collisions

Idea:

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- To search for key k:
    - Compute h(k) and examine/probe slot h(k).
    - If slot h(k) contains key k, the search is successful. If this slot contains NIL, the search is unsuccessful.

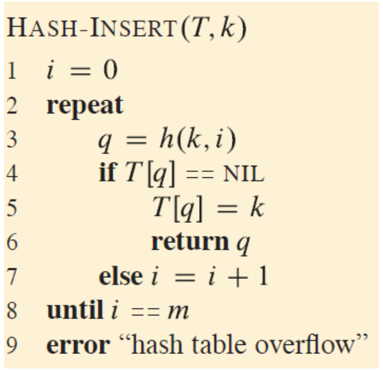
      There's a third possibility: slot h(k) contains a key that is not k. We compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on.
    - Keep probing until we either find key k (successful search) or we find a slot holding NIL (unsuccessful search).
- We need the sequence of slots probed to be a **permutation of the slot numbers** $\langle \mathbf{0}, \mathbf{1}, \ldots, \boldsymbol{m} - \mathbf{1} \rangle$ (so that we examine all slots if we must, and so that we don't examine any slot more than once).

Open addressing (2)

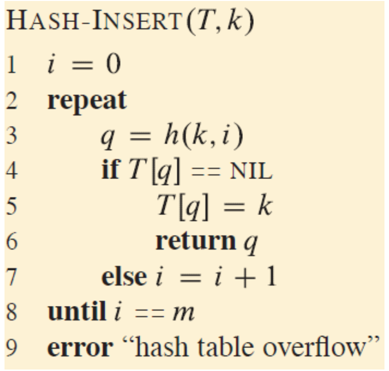
- Thus, the hash function is

$$h: U \times \underbrace{\{0,1,\dots,m-1\}}_{\text{Probe number}} \rightarrow \underbrace{\{0,1,\dots,m-1\}}_{\text{slot number}}$$

- The requirement that the sequence of slots be a permutation of $\langle \mathbf{0}, \mathbf{1}, \dots, \boldsymbol{m} - \mathbf{1} \rangle$ is equivalent to requiring that the probe sequence $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ be a permutation of $\langle \mathbf{0}, \mathbf{1}, \dots, \boldsymbol{m} - \mathbf{1} \rangle$.
- To insert, act as though we're searching, and insert at the first NIL slot we find.



```
HASH-INSERT(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error "hash table overflow"
```



```
HASH-INSERT(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error "hash table overflow"
```

Linear probing

- Given auxiliary hash function h',
  - the probe sequence starts at slot $h'(k)$ and
  - continues sequentially through the table, wrapping after slot $m-1$ to slot 0.
- Given key k and probe number i $(0 \leq i < m)$,

$$h(k,i) = (h'(k) + i) \bmod m$$

- The initial probe determines the entire sequence → only m possible sequences.
- Linear probing suffers from **primary clustering**:
  - long runs of occupied sequences build up and
  - long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
- Result is that the average search and insertion times increase.

Quadratic probing

- As in linear probing, the probe sequence starts at $h'(k)$
- Unlike linear probing, it jumps around in the table according to a quadratic function of the probe number:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- where are $c_1, c_2 \neq 0$ constants
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle \mathbf{0}, \mathbf{1}, \dots, \boldsymbol{m} - \mathbf{1} \rangle$
- Can result in **secondary clustering**:
    - if two distinct keys have the same initial hash h' , then
        - they have the same probe sequence.

Double hashing

- Use two auxiliary hash functions, h_1 and h_2.
  - h_1 gives the initial probe, and
  - h_2 gives the remaining probes:

$$h(k,i) = \big(h_1(k) + ih_2(k)\big) \bmod m$$

- Must have $h_2(k)$ to be relatively prime to m (no factors in common other than 1) to guarantee that the probe sequence is a full permutation of $\langle \mathbf{0}, \mathbf{1}, \dots, \boldsymbol{m} - \mathbf{1} \rangle$.
  - Could choose m to be a power of 2 and h_2 to always produce an odd number > 1.
  - Could let m be prime and have $1 < h_2(k) < m$.

- $\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence.

Double hash functions

- A good $h_2(k)$ should generate values that are relatively prime to the hash table size m.
    - **Relatively prime**: if two real numbers do not share a positive common divisor, besides 1.

- This can be achieved easily by defining m as a power of 2, i.e., $m = 2^p$ and ensure that $h_2(k)$ always returns an odd number.

- Alternatively, choose the double functions as:
    - $h_1(k) = k \bmod m$
    - $h_2(k) = 1 + (k \bmod m')$
- where m is a prime and m' is slightly smaller, e.g., $m' = m - 1$.

Summary

- The choice of data structure impacts the efficiency of algorithms => knowledge crucial to the design of algorithms
- **Hash table**:
    - Array with linked lists
    - A key maps to a slot in array (by hash function)
    - Collisions are avoided using chaining
- **Hash functions**:
    - **Division method**: Quick, but strict requirements to number of slots m for good performance.
    - **Multiplication method**: m can be chosen freely, but constant A can affect performance.
    - **Universal hash function**: prevents malicious provocation of worst-case search time by randomly choosing hash functions.
- **Open addressing**:
    - Data is stored directly in hash table array; probing to resolve collisions.
    - Clustering occurs with linear and quadratic methods; double hashing is better (but slower).