DATA STRUCTURES AND ALGORITHMS

LECTURE 9 - 10

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 8...

- ADT Matrix
- ADT List
- ADT Stack

Today

1. ADT Stack
2. ADT Queue
3. ADT Deque
4. ADT Priority Queue
5. Different problems
6. Hash tables

Stack - reminder

- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added at the top.
  - When an element is removed it will be removed automatically from the top.
  - Only the element from the top can be accessed.

- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

Stack - reminder

- Main operations for a Stack:
  - init, destroy
  - push, pop, top
  - isEmpty (maybe isFull)

- Possible representations for a Stack:
  - Static array
  - Dynamic array
  - Singly linked list
  - Doubly linked list

Singly-Linked List-based representation

?Where should we place the top of the stack for optimal performance?

Singly-Linked List-based representation

?Where should we place the top of the stack for optimal performance?

- We have two options:
  - Place it at the end of the list (like we did when we used an array) - for every push, pop and top operation we have to iterate through every element to get to the end of the list.
  - Place it at the beginning of the list - we can push and pop elements without iterating through the list.

Singly-Linked List-based representation

<u>Node:</u>
elem: TElem
next: ↑ Node

<u>Stack</u>
top: ↑ Node

Init - Implementation using a singly-linked list

subalgorithm init(s) **is:**
  s.top $\leftarrow$ NIL
end-subalgorithm

- Complexity: $\Theta(1)$

Destroy - Implementation using a singly-linked list

subalgorithm destroy(s) **is:**
    while s.top $\neq$ NIL **execute**
        firstNode $\leftarrow$ s.top
        s.top $\leftarrow$ [s.top].next
        @deallocate firstNode
    end-while
end-subalgorithm

- Complexity: $\Theta(n)$ - where n is the number of elements from s

Push - Implementation using a singly-linked list

subalgorithm push(s, e) **is:**
    //allocate a new Node and set its fields
    @allocate *newnode* of type Node
    [newnode].elem ← e
    [newnode].next ← NIL
    if s.top = NIL **then**
        s.top ← newnode
    else
        [newnode].next ← s.top
        s.top ← newnode
    end-if
end-subalgorithm

- Complexity: $\Theta(1)$

Pop - Implementation using a singly-linked list

function pop(s) **is:**
  if s.top = NIL **then** *//check if s is empty*
    @throw underflow(empty stack) exception
  end-if
  firstNode ←s.top
  topElem ← [firstNode].elem
  s.top ← [s.top].next
  @deallocate firstNode
  pop← topElem
end-function

- Complexity: $\Theta(1)$

Top - Implementation using a singly-linked list

function top(s) **is:**
  if s.top = NIL **then** *//check if s is empty*
    @throw underflow(empty stack) exception
  end-if
  topElem ← [s.top].elem
  top← topElem
end-function

- Complexity: $\Theta(1)$

IsEmpty - Implementation using a singly-linked list

function isEmpty(s) **is:**
  if s.top $=$ NIL **then**
    isEmpty $\leftarrow$ True
  else
    isEmpty $\leftarrow$ False
  end-if
end-function

- Complexity: $\Theta(1)$

IsFull - Implementation using a singly-linked list

- We don't have a maximum capacity in case of a linked list, so our stack will never be full. If we still want to implement this method, we can make it to always return false.

function isFull(s) **is:**
  isFull $\leftarrow$ False
end-function

- Complexity: $\Theta(1)$

Fixed capacity stack with singly-linked list

? How could we implement a stack with a fixed maximum capacity using a singly-linked list?

Fixed capacity stack with singly-linked list

? How could we implement a stack with a fixed maximum capacity using a singly-linked list?

- Similar to the implementation with a static array, we can keep in the *Stack* structure two integer values: maximum capacity and current size.

GetMinimum in constant time

? How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

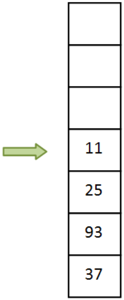
GetMinimum in constant time

? How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

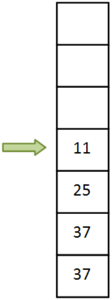
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time - Example

- If this is the *element stack*:



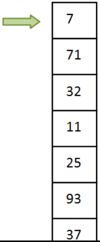
- This is the corresponding *min stack*:



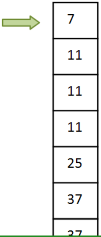
GetMinimum in constant time - Example

- When a new element is pushed to the *element stack*, we push a new element to the *min stack* as well. This element is the minimum between the top of the *min stack* and the newly added element.

- The *element stack*:



- The corresponding *min stack*:



GetMinimum in constant time

- When an element si popped from the *element stack*, we will pop an element from the *min stack* as well.
- The *getMinimum* operation will simply return the *top* of the *min stack*.
- The other stack operations remain unchanged (except *init*, where you have to create two stacks).

GetMinimum in constant time

- Let's implement the *push*, *pop* and *getMinimum* operations for this *SpecialStack*, represented in the following way:

SpecialStack:
  elementStack: Stack
  minStack: Stack

- We will use an existing implementation for the stack and work only with the operations from the interface.

Push for SpecialStack

```
subalgorithm push(ss, e) is:
  if isFull(ss.elementStack) then
    @throw overflow (full stack) exception
  end-if
  if isEmpty(ss.elementStack) then//the stacks are empty, just push the elem
    push(ss.elementStack, e)
    push(ss.minStack, e)
  else
    push(ss.elementStack, e)
    currentMin ← top(ss.minStack)
    if currentMin < e then //find the minim to push to minStack
      push(ss.minStack, currentMin)
    else
      push(ss.minStack, e)
    end-if
  end-if
end-subalgorithm //Complexity: Θ(1)
```

Pop for SpecialStack

function pop(ss) **is:**
  if isEmpty(ss.elementStack) **then**
    @throw underflow (empty stack) exception
  end-if
  currentElem $\leftarrow$ pop(ss.elementStack)
  pop(ss.minStack) *//we don't need the value, just to pop it*
  pop $\leftarrow$ currentElem
end-function

- Complexity: $\Theta(1)$

GetMinimum for SpecialStack

function getMinimum(ss) **is:**
  if isEmpty(ss.elementStack) **then**
    @throw underflow (empty stack) exception
  end-if
  getMinimum $\leftarrow$ top(ss.minStack)
end-function

- Complexity: $\Theta(1)$

SpecialStack - Notes / Think about it

- We designed the special stack in such a way that all the operations have a $\Theta(1)$ time complexity.
- The disadvantage is that we occupy twice as much space as with the regular stack.

? Think about how can we reduce the space occupied by the *min stack* to O(n) (especially if the minimum element of the stack rarely changes). *Hint: If the minimum does not change, we don't have to push a new element to the min stack.* How can we implement the *push* and *pop* operations in this case? What happens if the minimum element appears more than once in the *element stack*?

Delimiter matching

- Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
  - The sequence ()([][][(())]) - is correct
  - The sequence [()()()()] - is correct
  - The sequence [()]) - is not correct (one extra closed round bracket at the end)
  - The sequence [(]) - is not correct (brackets closed in wrong order)
  - The sequence {[[]] () - is not correct (curly bracket is not closed)

Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

Bracket matching - Implementation

```
function bracketMatching(seq) is:
   init(st) //create a stack
   for elem in seq execute
      if @ elem is open bracket then
         push(st, elem)
      else //elem is a closed bracket
         if isEmpty(st) then
            bracketMatching ← False //no open bracket at all
         else
            lastOpenedBracket ←pop(st)
            if not @lastOpenedBracket matches elem then
               bracketMatching ← False
            end-if
         end-if
      end-if
   end-for //continued on next slide...
```

Bracket matching - Implementation

if isEmpty(st) **then**
  bracketMatching ← True
else *//we have extra open bracket(s)*
  bracketMatching ← False
end-if
end-function

- Complexity: $\Theta(n)$ - where n is the length of the sequence

Bracket matching - Extension

- How can we extend the previous implementation so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch

Bracket matching - Extension

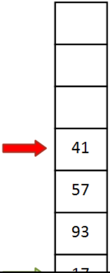
- How can we extend the previous implementation so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the sequence, and push to the stack $< delimiter, position >$ pairs.

ADT Queue

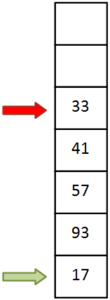
- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
  - When a new element is added (pushed), it has to be added to the *rear* of the queue.
  - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

ADT Queue - Example

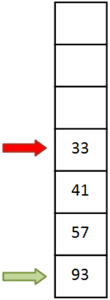
- Assume that we have this queue (green arrow is the front, red arrow is the rear)



- Push number 33:

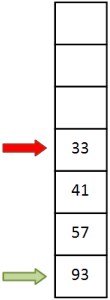


- Pop an element:

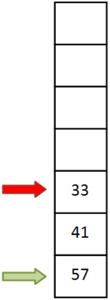


ADT Queue - Example

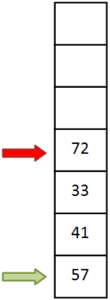
- This is our queue:



- Pop an element:



- Push number 72:



ADT Queue - Interface I

- The domain of the ADT Queue:
  $\mathcal{Q} = \{q | q \textit{ is a queue with elements of type TElem}\}$

- The interface of the ADT Queue contains the following operations:

ADT Queue - Interface II

- init(q)
  - **Description:** creates a new empty queue
  - **Pre:** True
  - **Post:** $q \in \mathcal{Q}$, q is an empty queue

ADT Queue - Interface III

- destroy(q)
  - **Description:** destroys a queue
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** q was destroyed

ADT Queue - Interface IV

- push(q, e)
  - **Description:** pushes (adds) a new element to the rear of the queue
  - **Pre:** $q \in \mathcal{Q}$, e is a $TElem$
  - **Post:** $q' \in \mathcal{Q}$, $q' = q \oplus e$, e is the element at the rear of the queue
  - **Throws:** an *overflow* error if the queue is full

ADT Queue - Interface V

- pop(q)
  - **Description:** pops (removes) the element from the front of the queue
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** $pop \leftarrow e$, e is a $TElem$, e is the element at the front of q, $q' \in \mathcal{Q}$, $q' = q \ominus e$
  - **Throws:** an *underflow* error if the queue is empty

ADT Queue - Interface VI

- top(q)
  - **Description:** returns the element from the front of the queue (but it does not change the queue)
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** $top \leftarrow e$, e is a $TElem$, e is the element from the front of q
  - **Throws:** an *underflow* error if the queue is empty

ADT Queue - Interface VII

- isEmpty(s)
  - **Description:** checks if the queue is empty (has no elements)
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:**

$$isEmpty \leftarrow \begin{cases} true, & if\ q\ has\ no\ elements \\ false, & otherwise \end{cases}$$

ADT Queue - Interface VIII

- isFull(q)
  - **Description:** checks if the queue is full - not every implementation has this operation
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:**

$$isFull \leftarrow \begin{cases} true, & if\ q\ is\ full \\ false, & otherwise \end{cases}$$

ADT Queue - Interface IX

- **Note:** queues cannot be iterated, so they don't have an *iterator* operation!

Queue - Representation

- What data structures can be used to implement a Queue?
    - Static Array
    - Dynamic Array
    - Singly Linked List
    - Doubly Linked List
- For each possible representation we will discuss where we should place the *front* and the *rear* of the queue and the complexity of the operations.

Queue - Array-based representation

- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?

Queue - Array-based representation

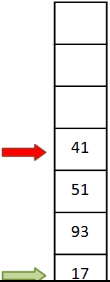
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

Queue - Array-based representation

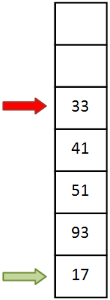
- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

Queue - Array-based representation

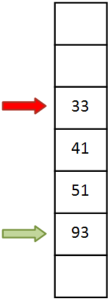
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:

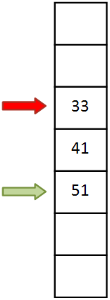


- Pop an element (and do not move the other elements):

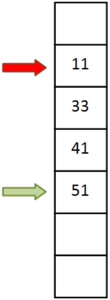


Queue - Array-based representation

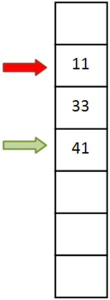
- Pop another element:



- Push number 11:

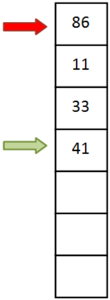


- Pop an element:

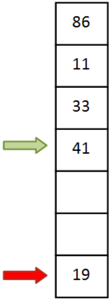


Queue - Array-based representation

- Push number 86:



- Push number 19:



Queue - representation on a circular array

- How can we represent a Queue on a circular array?

Queue:
  capacity: Integer
  front: Integer
  rear: Integer
  elems: TElem[]

- Optionally, the *length* of the queue could also be kept as a part of the structure.
- Front and rear (in this implementation) are positions actually occupied by the elements.

Queue - representation on a circular array - init

- We will use the value -1 for *front* and *end*, to denote an empty queue.

subalgorithm init(q) **is:**
  q.capacity ← INIT_CAPACITY //*some constant*
  q.front ← -1
  q.rear ← -1
  @allocate memory for the *elems* array
end-subalgorithm

- Complexity: $\Theta(1)$

Queue - representation on a circular array - isEmpty

- How do we check whether the queue is empty?

function isEmpty(q) **is:**
  if q.front = -1 **then**
    isEmpty $\leftarrow$ True
  else
    isEmpty $\leftarrow$ False
  end-if
end-function

- Complexity: $\Theta(1)$

Queue - representation on a circular array - top

- What should the *top* operation do?

function top(q) **is:**
  if q.front != -1 **then**
    top ← q.elems[q.front]
  else
    @error - queue is empty
  end-if
end-function

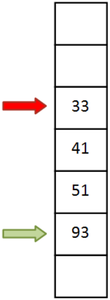
- Complexity: $\Theta(1)$

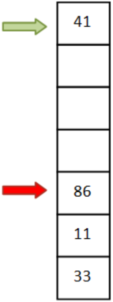
Queue - representation on a circular array - pop

- What should the *pop* operation do?

Queue - representation on a circular array - pop

- There are two situations for our queue:





Queue - representation on a circular array - pop

```
function pop (q) is:
    if q.front != -1 then
        deletedElem ← q.elems[q.front]
        if q.front = q.rear then //we have one single element
            q.front ← -1
            q.rear ← -1
        else if q.front = q.cap then
            q.front ← 1
        else
            q.front ← q.front + 1
        end-if
        pop ← deletedElem
    end-if
    @error - queue is empty
end-function
```

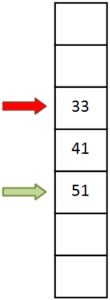
- Complexity: $\Theta(1)$

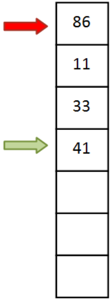
Queue - representation on a circular array - push

- What should the *push* operation do?

Queue - representation on a circular array - push

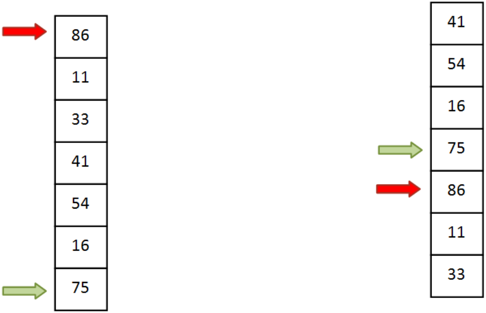
- There are two situations for our queue:





Queue - representation on a circular array - push

- When pushing a new element we have to check whether the queue is full



- For both example, the elements were added in the order: 75,

Queue - representation on a circular array - push

- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)
- When the existing elements are copied, we have to *straighten out* the array.

Queue - representation on a circular array - push

subalgorithm push(q, e) **is:**
  if q.front = -1 **then**
    q.elems[1] ← e
    q.front ← 1
    q.rear ← 1
    @return
  else if(q.front=1 **and** q.rear=a.cap) **OR** q.rear=q.front-1 **then**
    @resize
  end-if
  if q.rear ≠ q.cap **then**
    q.elems[q.rear+1] ← e
    q.rear ← q.rear + 1
  else
    q.elems[1] ← e
    q.rear ← 1
  end-if
end-subalgorithm

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

Queue - representation on a SLL

- We can improve the complexity of the operations if, even though the list is singly linked, we keep both the head and the tail of the list.
- What should the tail of the list be: the *front* or the *rear* of the queue?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning
- In either case we will have both operations (push or pop) in $\Theta(1)$ complexity.

Evaluating an arithmetic expression

- We want to write an algorithm that can compute the result of an arithmetic expression:
- For example:
  - 2+3*4 = 14
  - ((2+4)*7)+3*(9-5) = 54
  - ((((3+1)*3)/((9-5)+2))-((3*(7-4)) + 6)) = -13

- An arithmetic expression is composed of *operators* (+, -, * or /), parentheses and *operands* (the numbers we are working with). For simplicity we are going to use single digits as operands and we suppose that the expression is correct.

Infix and postfix notations

- The arithmetic expressions presented on the previous slide are in the so-called *infix* notation. This means that the *operators* are between the two operands that they refer to. Humans usually use this notation, but for a computer algorithm it is complicated to compute the result of an expression in an infix notation.
- Computers can work a lot easier with the *postfix* notation, where the operator comes after the operands.

Infix and postfix notations

- Examples of expressions in infix notation and the corresponding postfix notations:

Infix notation	Postfix notation
1+2	12+
1+2-3	12+3-
4*3+6	43*6+
4*(3+6)	436+*
(5+6)*(4-1)	56+41-*
1+2*(3-4/(5+6))	123456+/-*+

- The order of the operands is the same for both the infix and the postfix notations, only the order of the operators changes
- The operators have to be ordered taking into consideration

Infix and postfix notations

- So, evaluating an arithmetic expression is divided into two subproblems:
  - Transform the infix notation into a postfix notation
  - Evaluate the postfix notation
- Both subproblems are solved using stacks and queues.

Infix to postfix transformation - The main idea

- Use an auxiliary stack for the operators and parentheses and a queue for the result.
- Start parsing the expression.
- If an operand is found, push it to the queue
- If an open parenthesis is found, it is pushed to the stack.
- If a closed parenthesis is found, pop elements from the stack and push them to the queue until an open parenthesis is found (but do not push parentheses to the queue).

Infix to postfix transformation - The main idea

- If an operator (opCurrent) is found:
  - If the stack is empty, push the operator to the stack
  - While the top of the stack contains an operator with a higher or equal precedence than the current operator, pop and push to the queue the operator from the stack. Push opCurrent to the stack when the stack becomes empty, its top is a parenthesis or an operator with lower precedence.
  - If the top of the stack is open parenthesis or operator with lower precedence, push opCurrent to the stack.
- When the expression is completely parsed, pop everything from the stack and push to the queue.

Infix to postfix transformation - Example

- Let's follow the transformation of 1+2*(3-4/(5+6))+7

Infix to postfix transformation - Example

- Let's follow the transformation of 1+2*(3-4/(5+6))+7

Input	Operation	Stack	Queue
1	Push to Queue		1
+	Push to stack	+	1
2	Push to Queue	+	12
*	Check (no Pop) and Push	+*	12
(	Push to stack	+*(	12
3	Push to Queue	+*(	123
-	Check (no Pop) and Push	+*(-	123
4	Push to Queue	+*(-	1234
/	Check (no Pop) and Push	+*(-/	1234
(	Push to stack	+*(-/(	1234
5	Push to Queue	+*(-/(	12345
+	Check (no Pop) and Push	+*(-/(+	12345
6	Push to Queue	+*(-/(+	123456
)	Pop and push to Queue till (	+*(-/	123456+
)	Pop and push to Queue till (	+*	123456+/-
+	Check, Pop twice and Push	+	123456+/-*+
7	Push to Queue	+	123456+/-*+7
over	Pop everything and push to Queue		123456+/-*+7+

Infix to postfix transformation - Implementation

function infixToPostfix(expr) **is:**
    init(st)
    init(q)
    for elem **in** expr **execute**
        if @elem is an operand **then**
            push(q, elem)
        else if @ elem is open parenthesis **then**
            push(st, elem)
        else if @elem is a closed parenthesis **then**
            while @ top(st) is not an open parenthesis **execute**
                op ← pop(st)
                push(q, op)
            end-while
            pop(st) *//get rid of open parenthesis*
        else *//we have operand*
//continued on the next slide

Infix to postfix transformation - Implementation

```
        while not isEmpty(st) and @ top(st) not open parenthesis and @
top(st) has >= precedence than elem execute
            op ← pop(st)
            push(q, op)
        end-while
        push(st, elem)
    end-if
  end-for
  while not isEmpty(st) execute
    op ← pop(st)
    push(q, op)
  end-while
  infixtoPostfix ← q
end-function
```

- Complexity: $\Theta(n)$ - where n is the length of the sequence

Evaluation of expression in postfix notation

- Once we have the postfix notation we can compute the value of the expression using a stack
- The main idea of the algorithm:
    - Use an auxiliary stack
    - Start parsing the expression
    - If an operand if found, it is pushed to the stack
    - If an operator is found, two values are popped from the stack, the operation is performed and the result is pushed to the stack
    - When the expression is parsed, the stack contains the result

Evaluation of postfix notation - Example

- Let's follow the evaluation of 123456+/-*+7+

Evaluation of postfix notation - Example

- Let's follow the evaluation of 123456+/-*+7+

Pop from the queue	Operation	Stack
1	Push	1
2	Push	1 2
3	Push	1 2 3
4	Push	1 2 3 4
5	Push	1 2 3 4 5
6	Push	1 2 3 4 5 6
+	Pop, add, Push	1 2 3 4 11
/	Pop, divide, Push	1 2 3 0
-	Pop, subtract, Push	1 2 3
*	Pop, multiply, Push	1 6
+	Pop, add, Push	7
7	Push	7 7
+	Pop, add, Push	14

Evaluation of postfix notation - Implementation

```
function evaluatePostfix(q) is:
   init(st)
   while not isEmpty(q) execute
      elem ← pop(q)
      if @ elem is an operand then
         push(st, elem)
      else
         op1 ← pop(st)
         op2 ← pop(st)
         @ compute the result of op2 elem op1 in variable result
         push(st, result)
      end-if
   end-while
   result ← pop(st)
   evaluatePostfix ← result
end-function
```

Evaluation of an arithmetic expression

- Combining the two functions we can compute the result of an arithmetic expression.
- How can we evaluate directly the expression in infix notation with one single function? *Hint: use two stacks.*
- How can we add exponentiation as a fifth operation?

ADT Deque

- The ADT Deque (Double Ended Queue) is a container in which we can insert and delete from both ends:
  - We have *push_front* and *push_back*
  - We have *pop_front* and *pop_back*
  - We have *top_front* and *top_back*
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

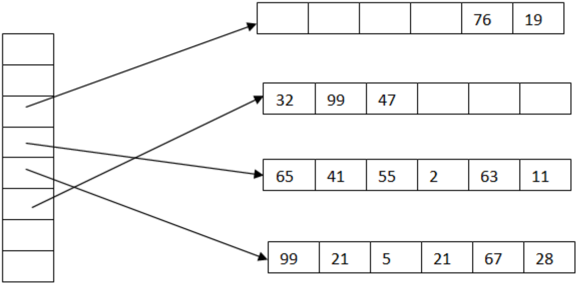
ADT Deque

- Possible representations for a Deque:
  - Circular Array
  - Doubly Linked List
  - A dynamic array of constant size arrays

ADT Deque - Representation

- An interesting representation for a deque is to use a dynamic array of fixed size arrays:
  - Place the elements in fixed size arrays (blocks).
  - Keep a dynamic array with the addresses of these blocks.
  - Every block is full, except for the first and last ones.
  - The first block is filled from right to left.
  - The last block is filled from left to right.
  - If the first or last block is full, a new one is created and its address is put in the dynamic array.
  - If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved).

Deque - Example



- Elements of the deque: 76, 19, 65, ..., 11, 99, ..., 28, 32, 99, 47

Deque - Example

- Information (fields) we need to represent a deque using a dynamic array of blocks:
  - Block size
  - The dynamic array with the addresses of the blocks
  - Capacity of the dynamic array
  - First occupied position in the dynamic array
  - First occupied position in the first block
  - Last occupied position in the dynamic array
  - Last occupied position in the last block
  - The last two fields are not mandatory if we keep count of the total number of elements in the deque.

ADT Priority Queue

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

ADT Priority Queue

- In order to work in a more general manner, we can define a relation $\mathcal{R}$ on the set of priorities: $\mathcal{R} : TPriority \times TPriority$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation $\mathcal{R}$.
- If the relation $\mathcal{R} = "\geq"$, the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathcal{R} = "\leq"$, the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

Priority Queue - Interface I

- The domain of the ADT Priority Queue:
  $\mathcal{PQ} = \{pq | pq$ is a priority queue with elements $(e, p), e \in TElem, p \in TPriority\}$
- The interface of the ADT Priority Queue contains the following operations:

Priority Queue - Interface II

- init (pq, R)
  - **Description:** creates a new empty priority queue
  - **Pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **Post:** $pq \in \mathcal{PQ}$, pq is an empty priority queue

Priority Queue - Interface III

- destroy(pq)
  - **Description:** destroys a priority queue
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** pq was destroyed

Priority Queue - Interface IV

- push(pq, e, p)
  - **Description:** pushes (adds) a new element to the priority queue
  - **Pre:** $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **Post:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

Priority Queue - Interface V

- pop (pq, e, p)
  - **Description:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**$e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority. $pq' \in \mathcal{PQ}, pq' = pq \ominus (e, p)$
  - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VI

- top (pq, e, p)
  - **Description:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
  - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VII

- isEmpty(pq)
  - **Description:** checks if the priority queue is empty (it has no elements)
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**

$$isEmpty \leftarrow \begin{cases} true, & if \ \ pq \ \ has \ \ no \ \ elements \\ false, & otherwise \end{cases}$$

Priority Queue - Interface VIII

- isFull (pq)
  - **Description:** checks if the priority queue is full (not every implementation has this operation)
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**

$$isFull \leftarrow \begin{cases} true, & if\ pq\ is\ full \\ false, & otherwise \end{cases}$$

Priority Queue - Interface IX

- **Note:** priority queues cannot be iterated, so they don't have an *iterator* operation!

Priority Queue - Representation

- What data structures can be used to implement a priority queue?
  - Dynamic Array
  - Linked List
  - (Binary) Heap

Priority Queue - Representation

- If the representation is a Dynamic Array or a Linked List we have to decide how we store the elements in the array/list:
  - we can keep the elements ordered by their priorities
  - we can keep the elements in the order in which they were inserted

Priority Queue - Representation

- Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	O(n)	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

- What happens if we keep in a separate field the element with the highest priority?

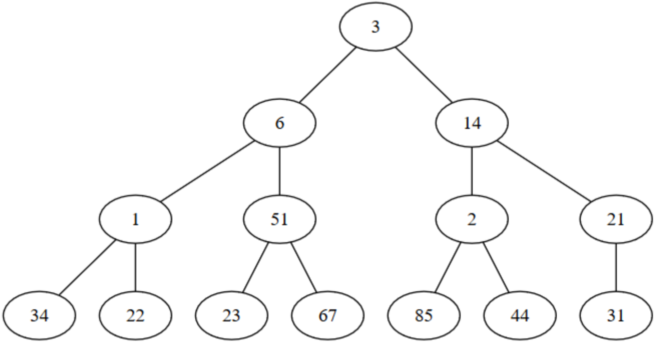
Priority Queue - Representation

- Another representation for a Priority Queue is to use a binary heap, where the root of the heap is the element with the highest priority (the figure contains the priorities only).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31

Priority Queue - Representation

- The previous array, interpreted as a binary heap:



Priority Queue - Representation on a binary heap

- When an element is pushed to the priority queue, it is simply added to the heap (and bubbled-up if needed)
- When an element is popped from the priority queue, the root is removed from the heap (and bubble-down is performed if needed)
- Top simply returns the root of the heap.

Priority Queue - Representation

- Let's complete our table with the complexity of the operations if we use a heap as representation:

Operation	Sorted	Non-sorted	Heap
push	O(n)	$\Theta(1)$	$O(log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

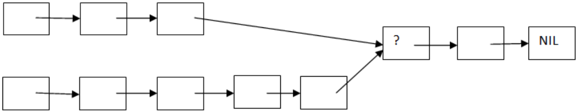
- Consider the total complexity of the following sequence of operations:
  - start with an empty priority queue
  - push n random elements to the priority queue
  - perform pop n times

Priority Queue - Applications

- Problems where a priority queue can be used:
  - Triage procedure in the hospitals
  - Scholarship allocation - see Seminar 5
  - Give me a ticket on an airplane (war story from Steven S. Skiena: *The Algorithm Design Manual*, Second Edition, page 118)

Think about it - Linked Lists

- Write a non-recursive algorithm to reverse a singly linked list with $\Theta(n)$ time complexity, using constant space/memory.
- Suppose there are two singly linked lists both of which intersect at some point and become a single linked list (see the image below). The number of nodes in the two lists before the intersection is not known and may be different in each list. Give an algorithm for finding the merging point (hint - use a Stack)



Think about it - Stacks and Queues I

- How can we implement a Stack using two Queues? What will be the complexity of the operations?
- How can we implement a Queue using two Stacks? What will be the complexity of the operations?
- How can we implement two Stacks using only one array? The stack operations should throw an exception only if the total number of elements in the two Stacks is equal to the size of the array.

Think about it - Stacks and Queues II

- Given a string of lower-case characters, recursively remove adjacent duplicate characters from the string. For example, for the word "mississippi" the result should be "m".
- Given an integer k and a queue of integer numbers, how can we reverse the order of the first k elements from the queue? For example, if $k=4$ and the queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90], the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

Think about it - Priority Queues

- How can we implement a stack using a Priority Queue?
- How can we implement a queue using a Priority Queue?

Different problems I

- Red-Black Card Game:
  - Statement: Two players each receive $\frac{n}{2}$ cards, where each card can be red or black. The two players take turns; at every turn the current player puts the card from the upper part of his/her deck on the table. If a player puts a red card on the table, the other player has to take all cards from the table and place them at the bottom of his/her deck. The winner is the player that has all the cards.
  - Requirement: Given the number n of cards, simulate the game and determine the winner.
  - Hint: use stack(s) and queue(s)

Different problems II

- Robot in a maze:
  - Statement: There is a rectangular maze, composed of occupied cells (X) and free cells (*). There is a robot (R) in this maze and it can move in 4 directions: N, S, E, V.
  - Requirements:
    - Check whether the robot can get out of the maze (get to the first or last line or the first or last column).
    - Find a path that will take the robot out of the maze (if exists).

```
X * * X X X * *
X * X * * * * *
X * * * * * X *
X X X * * * X *
* X * * R X X *
* * * X X X X *
* * * * * * * X
X X X X X X X X
```

Different problems III

- Hint - Version 1:
  - Let T be the set of positions where the robot can get from the starting position.
  - Let S be the set of positions to which the robot can get at a given moment and from which it could continue going to other positions.

Different problems IV

- A possible way of determining the sets T and S could be the following:

T $\leftarrow$ {initial position}
S $\leftarrow$ {initial position}
while S $\neq \emptyset$ **execute**
    Let p be one element of S
    S $\leftarrow$ S $\setminus$ $\{p\}$
    for each valid position q where we can get from p and which is not in T **do**
        T $\leftarrow$ T $\cup$ $\{q\}$
        S $\leftarrow$ S $\cup$ $\{q\}$
    end-for
end-while

- T can be a list, a vector or a matrix associated to the maze

Different problems V

- S can be a stack or a queue (or even a priority queue, depending on what we want)

Different problems VI

- Hint - Version 2:
  - The solution is similar to the one presented on the previous slide.
  - If S is a queue, and T is a stack extended with the search operation, once we got out of the maze, T can be used to build the list of positions that got us to the margin of the maze. In this case we need both a stack and a queue.

Different problems VII

- How can we merge k sorted singly linked lists? How can we do it in $O(n * log_2 k)$ complexity (n is the total number of elements from the k lists)?

Direct-address tables I

- Consider the following problem:
  - We have data where every element has a key (a natural number).
  - The universe of keys (the possible values for the keys) is relatively small, $U = \{0, 1, 2, \ldots, m-1\}$
  - No two elements have the same key
  - We have to support the basic dictionary operations: INSERT, DELETE and SEARCH

Direct-address tables II

- Example 1: Store data about different bus lines for a city's public transportation service
    - We can consider the bus number as a key, and the data to be stored as a value (satellite data)
    - The bus numbers belong to a relatively small interval - in Cluj-Napoca it is around 100
    - Bus numbers are unique
- Example 2: Store data about students based on their registration numbers (a number from the 1 - 9999 interval, but there may be unused numbers - students that left the university)

Direct-address tables III

- Solution:
  - Use an array T with m positions (remember, the keys belong to the $[0, m-1]$ interval)
  - Data about element with key k, will be stored in the $T[k]$ slot
  - Slots not corresponding to existing elements will contain the value NIL (or some other special value to show that they are empty)

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
    search ← T[k]
end-function

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
  search ← T[k]
end-function

subalgorithm insert(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
  T[key(x)] ← x *//key(x) returns the key of an element*
end-subalgorithm

Operations for a direct-address table

function search(T, k) **is**:
//pre: T is an array (the direct-address table), k is a key
  search ← T[k]
end-function

subalgorithm insert(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
  T[key(x)] ← x *//key(x) returns the key of an element*
end-subalgorithm

subalgorithm delete(T, x) **is**:
//pre: T is an array (the direct-address table), x is an element
  T[key(x)] ← NIL
end-subalgorithm

Direct-address table - Advantages and disadvantages

- Advantages of direct address-tables:
  - They are simple
  - They are efficient - all operations run in $\Theta(1)$ time.
- Disadvantages of direct address-tables - restrictions:
  - The keys have to be natural numbers
  - The keys have to come from a small universe (interval)
  - The number of actual keys can be a lot less than the cardinal of the universe (storage space is wasted)

Hash tables

- Hash tables are generalizations of direct-address tables and they represent a *time-space trade-off*.
- Searching for an element still takes $\Theta(1)$ time, but as *average case complexity* (worst case complexity is higher)

Hash tables - main idea I

- We will still have a table T of size m (but now m is not the number of possible keys, $|U|$) - *hash table*
- Use a function h that will map a key k to a slot in the table T - *hash function*

$$h : U \rightarrow \{0, 1, ..., m - 1\}$$

- Remarks:
  - In case of direct-address tables, an element with key k is stored in $T[k]$.
  - In case of hash tables, an element with key k is stored in $T[h(k)]$.

Hash tables - main idea II

- The point of the hash function is to reduce the range of array indexes that need to be handled $=>$ instead of $|U|$ values, we only need to handle m values.
- Consequence:
  - two keys may hash to the same slot $=>$ **a collision**
  - we need techniques for resolving the conflict created by collisions

A good hash function I

- A good hash function:
  - can minimize the number of collisions (but cannot eliminate all collisions)
  - is deterministic
  - can be computed in $\Theta(1)$ time

A good hash function II

- satisfies (approximately) the assumption of simple uniform hashing: **each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to**

$$P(h(k) = j) = \frac{1}{m} \; \forall j = 0, ..., m-1 \;\; \forall k \in U$$

- Examples of bad hash functions:
  - $h(k) =$ constant number
  - $h(k) =$ random number
  - $h(k) = k \; mod \; 10$ - when $m > 10$
  - etc.

A good hash function III

- In practice we use heuristic techniques to create hash functions that perform well.
- Most hash functions assume that the keys are natural numbers. If this is not true, they have to be interpreted as natural number. In what follows, we assume that the keys are natural numbers.
- There are different methods of defining a hash function:
  - The division method
  - The multiplication method
  - Universal hashing

The division method

The division method

$h(k) = k \quad mod \quad m$

For example:

$$\text{m} = 13$$
$$\text{k} = 63 => \text{h(k)} = 11$$
$$\text{k} = 52 => \text{h(k)} = 0$$
$$\text{k} = 131 => \text{h(k)} = 1$$

- Requires only a division so it is quite fast
- Experiments show that good values for m are primes not too close to exact powers of 2

The multiplication method I

The multiplication method

$$h(k) = floor(m * frac(k * A)) \text{ where}$$

m - the hash table size

A - constant in the range $0 < A < 1$

$frac(k * A)$ - fractional part of $k * A$

For example

m = 13 A = 0.6180339887

k=63 => h(k) = floor(13 * frac(63 * A)) = floor(12.16984) = 12

k=52 => h(k) = floor(13 * frac(52 * A)) = floor(1.790976) = 1

k=129=> h(k)= floor(13 * frac(129 * A)) = floor(9.442999) = 9

The multiplication method II

- Advantage: the value of m is not critical, typically $m = 2^p$ for some integer p
- Some values for A work better than others. Knuth suggests $\frac{\sqrt{5}-1}{2} = 0.6180339887$

Universal hashing I

- If we know the exact hash function used by a hash table, we can always generate a set of keys that will hash to the same position (collision). This reduces the performance of the table.
- For example:

$m = 13$
$h(k) = k \;\; mod \;\; m$
k = 11, 24, 37, 50, 63, 76, etc.

Universal hashing II

- Instead of having one hash function, we have a collection $\mathcal{H}$ of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$
- Such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$ the number of hash functions from $\mathcal{H}$ for which $h(x) = h(y)$ is precisely $\frac{|\mathcal{H}|}{m}$
- In other words, with a hash function randomly chosen from $\mathcal{H}$ the chance of collision between x and y, where $x \neq y$, is exactly $\frac{1}{m}$

Universal hashing III

Example 1

Fix a prime number $p >$ *the maximum possible value for a key from* U.
For every $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we can define a hash function $h_{a,b}(k) = ((a * k + b) \ mod \ p) \ mod \ m$.

- For example:
  - $h_{3,7}(k) = ((3 * k + 7) \ mod \ p) \ mod \ m$
  - $h_{4,1}(k) = ((4 * k + 1) \ mod \ p) \ mod \ m$
  - $h_{8,0}(k) = ((8 * k) \ mod \ p) \ mod \ m$
- There are $p * (p-1)$ possible hash functions that can be chosen.

Universal hashing IV

Example 2

If the key k is an array $< k_1, k_2, \ldots, k_r >$ such that $k_i < m$ (or it can be transformed into such an array).
Let $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$.
$h(k) = \sum_{i=1}^{r} k_i * x_i \;\; mod \; m$

Universal hashing V

Example 3

Suppose the keys are $u - bits$ long and $m = 2^b$.
Pick a random $b - by - u$ matrix (called h) with 0 and 1 values only.
Pick $h(k) = h * k$ where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Using keys that are not natural numbers I

- The previously presented hash functions assume that keys are natural numbers.
- If this is not true there are two options:
  - Define special hash functions that work with your keys (for example, for real number from the [0,1) interval $h(k) = [k * m]$ can be used)
  - Use a function that transforms the key to a natural number (and use any of the above-mentioned hash functions) - *hashCode*

Using keys that are not natural numbers II

- If the key is a string s:
  - we can consider the ASCII codes for every letter
  - we can use 1 for *a*, 2 for *b*, etc.
- Possible implementations for *hashCode*
  - $s[0] + s[1] + ... + s[n-1]$
    - Anagrams have the same sum *SAUCE* and *CAUSE*
    - *DATES* has the same sum (D = C + 1, T = U - 1)
    - Assuming maximum length of 10 for a word (and the second letter representation), *hashCode* values range from 1 (the word *a*) to 260 (*zzzzzzzzzz*). Considering a dictionary of about 50,000 words, we would have on average 192 word for a *hashCode* value.

Using keys that are not natural numbers III

- $s[0] * 26^{n-1} + s[1] * 26^{n-2} + ... + s[n-1]$ where
  - n - the length of the string
  - Generates a much larger interval of *hashCode* values.
  - Instead of 26 (which was chosen since we have 26 letters) we can use a prime number as well (Java uses 31, for example).

Collisions

- When two keys, x and y, have the same value for the hash function $h(x) = h(y)$ we have a *collision*.
- A good hash function can reduce the number of collisions, but it cannot eliminate them at all:
  - Try fitting $m + 1$ keys into a table of size m
- There are different collision resolution methods:
  - Separate chaining
  - Coalesced chaining
  - Open addressing

The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).

The birthday paradox

- *How many randomly chosen people are needed in a room, to have a good probability - about 50% - of having two people with the same birthday?*
- It is obvious that if we have 367 people, there will be at least two with the same birthday (there are only 366 possibilities).
- What might not be obvious, is that approximately 70 people are needed for a 99.9% probability
- 23 people are enough for a 50% probability

Separate chaining

- Collision resolution by chaining: each slot from the hash table T contains a linked list, with the elements that hash to that slot
- Dictionary operations become operations on the corresponding linked list:
  - $insert(T, x)$ - insert a new node to the beginning of the list $T[h(key[x])]$
  - $search(T, k)$ - search for an element with key k in the list $T[h(k)]$
  - $delete(T, x)$ - delete x from the list $T[h(key[x])]$

Hash table with separate chaining - representation

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

<u>Node:</u>
key: TKey
next: ↑ Node

<u>HashTable:</u>
T: ↑Node[] *//an array of pointers to nodes*
m: Integer
h: TFunction *//the hash function*

Hash table with separate chaining - insert

subalgorithm insert(ht, k) **is**:
//pre: ht is a HashTable, k is a TKey
//post: k was inserted into ht
    position ← ht.h(k)
    allocate(newNode)
    [newNode].next ← NIL
    [newNode].key ← k
    if ht.T[position] = NIL **then**
        ht.T[position] ← newNode
    else
        [newNode].next ← ht.T[position]
        ht.T[position] ← newNode
    end-if
end-subalgorithm

Hash table with separate chaining - search

function search(ht, k) **is**:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentN ← ht.T[position]
    while currentN ≠ NIL **and** [currentN].key ≠ k **execute**
        currentN ← [currentN].next
    end-while
    if currentN ≠ NIL **then**
        search ← True
    else
        search ← False
    end-if
end-function

- Usually search returns the info associated with the key k

Analysis of hashing with chaining

- The average performance depends on how well the hash function h can distribute the keys to be stored among the m slots.
- **Simple Uniform Hashing** assumption: each element is equally likely to hash into any of the m slots, independently of where any other elements have hashed to.
- **load factor** α of the table T with m slots containing n elements
  - is n/m
  - represents the average number of elements stored in a chain
  - in case of separate chaining can be less than, equal to, or greater than 1.

Analysis of hashing with chaining - Insert

- The slot where the element is to be added can be:
    - empty - create a new node and add it to the slot
    - occupied - create a new node and add it to the beginning of the list
- In either case worst-case time complexity is: $\Theta(1)$
- If we have to check whether the element already exists in the table, the complexity of searching is added as well.

Analysis of hashing with chaining - Search I

- There are two cases
  - unsuccessful search
  - successful search
- We assume that
  - the hash value can be computed in constant time ($\Theta(1)$)
  - the time required to search an element with key k depends linearly on the length of the list $T[h(k)]$

Analysis of hashing with chaining - Search II

- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

- Proof idea: $\Theta(1)$ is needed to compute the value of the hash function and α is the average time needed to search one of the *m* lists

Analysis of hashing with chaining - Search III

- If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table)
  - $\alpha = n/m = O(m)/m = O(1)$
  - searching takes constant time on average
- Worst-case time complexity is $\Theta(n)$
  - When all the nodes are in a single linked-list and we are searching this list
  - In practice hash tables are pretty fast

Analysis of hashing with chaining - Delete

- If the lists are doubly-linked and we know the address of the node: $\Theta(1)$
- If the lists are singly-linked: proportional to the length of the list
- **All dictionary operations can be supported in $\Theta(1)$ time on average.**
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to α. If α is too large $\Rightarrow$ resize and rehash.