Algorithms and Data Structures
CS-CO-412

David Vernon
Professor of Informatics
University of Skövde
Sweden

david@vernon.eu
www.vernon.eu

Containers & Dictionaries

Lecture 6

Topic Overview

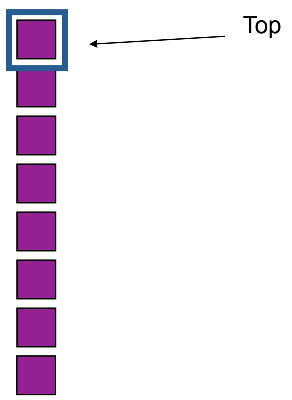
- Containers
- Dictionaries
- List ADT
  - Array implementation
  - Linked list implementation
- **Stack ADT**
- **Queue ADT**
- **Hashing**

Stacks & Queues

Stacks

- A stack is a special type of list
  - all insertions and deletions take place at one end, called the top
  - thus, the last one added is always the first one available for deletion
  - also referred to as
    - pushdown stack
    - pushdown list
    - LIFO list (Last In First Out)

Stacks



Stack Operations

- *Declare*: → **S** :

  The function value of *Declare(S)* is an empty stack

Stack Operations

- *Empty*: → **S** :

  The function *Empty* causes the stack to be emptied and it returns position *End(S)*

Stack Operations

- *IsEmpty*: $\mathbf{S} \rightarrow \mathbf{B}$:

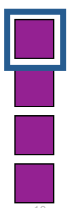
  The function value *IsEmpty*(*S*) is *true* if *S* is empty; otherwise it is *false*

Stack Operations

- *Top*: $\mathbf{S} \rightarrow \mathbf{E}$:

  The function value *Top*(*S*) is the first element in the list;

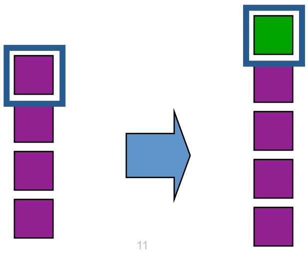
  if the list is empty, the value is undefined



Stack Operations

- *Push*: $\mathbf{E} \times \mathbf{S} \rightarrow \mathbf{L}$:

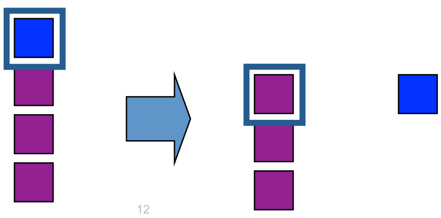
  Push(e, S)
  Insert an element *e* at the top of the stack



Stack Operations

- *Pop*: $\mathbf{S} \rightarrow \mathbf{E}$:

  Pop(S)
  Remove the top element from the stack: i.e. return the top element and delete it from the stack



Stack Operations

- *All these operations can be directly implemented using the LIST ADT operations on a List S*
- *Although it may be more efficient to use a dedicated implementation*
- *It depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Stack Operations

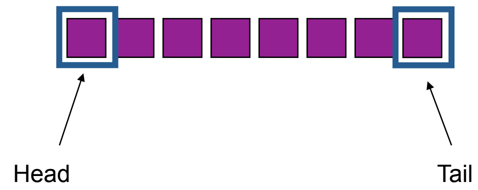
- Declare(S)
- Empty(S)
- Top(S)
  - Retrieve(First(S), S)
- Push(e, S)
  - Insert(e, First(S), S)
- Pop(S)
  - Retrieve(First(S), S)
  - Delete(First(S), S)

Queues

Queues

- A queue is another special type of list
  - insertions are made at one end, called the tail of the queue
  - deletions take place at the other end, called the head
  - thus, the last one added is always the last one available for deletion
  - also referred to as
    - FiFO list (First In First Out)

Queues



Queue Operations

- *Declare*: $\rightarrow$ **Q** :

  The function value of *Declare(Q)* is an empty queue

Queue Operations

- *Empty*: $\rightarrow$ **Q** :

  The function *Empty* causes the queue to be emptied and it returns position *End(Q)*

Queue Operations

- *IsEmpty*: **Q** $\rightarrow$ **B** :

  The function value *IsEmpty*(*Q*) is *true* if *Q* is empty; otherwise it is *false*

Queue Operations

- *Head*: **Q** → **E** :

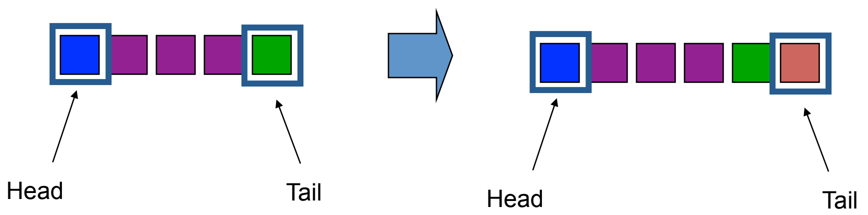
  The function value *Head*(*Q*) is the first element in the list;

  if the queue is empty, the value is undefined

Queue Operations

- *Enqueue*: **E** × **Q** → **Q** :

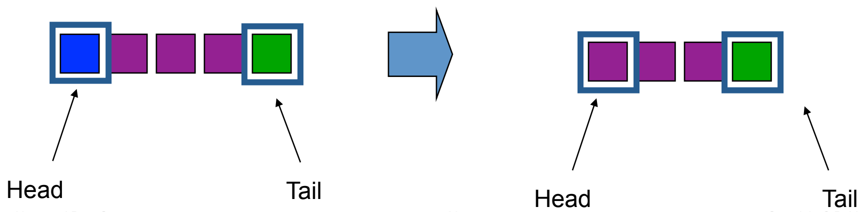
  Enqueue(e, Q)
  Add an element *e* the the tail of the queue



Queue Operations

- *Deqeue*: **Q** → **E** :

  Dequeue(Q)
  Remove the element from the head of the queue: i.e. return the first element and delete it from the queue



Queue Operations

- *All these operations can be directly implemented using the LIST ADT operations on a queue Q*
- *Again, it may be more efficient to use a dedicated implementation*
- *And, again, it depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Queue Operations

- Declare(Q)
- Empty(Q)
- Head(Q)
  - Retrieve(First(Q), Q)
- Enqueue(e, Q)
  - Insert(e, End(Q), Q)
- Dequeque(Q)
  - Retrieve(First(Q), Q)
  - Delete(First(Q), Q)

Hashing

- Hash tables are a **very** practical way to maintain a dictionary
- It takes a constant amount of time to look up an item in an array, **if** you have its index … O(1)
- A hash function is a mathematical function to maps keys to integers
  - This integer is used as an index into an array
  - We store our item at that position

Hashing

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Next place part number 6702 in the array.

Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to Resolve the Collision?

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

One way is by linear/sequential probing. This uses the rehash function

(HashValue + 1) % 100

repeatedly until an empty location is found for part number 6702.

Resolving the Collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

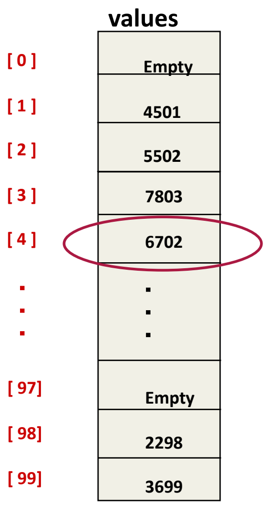
(HashValue + 1) % 100

Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision Resolved



Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Hashing

- In general, the keys are not so conveniently defined (e.g. part numbers) and they have to be computed
- Typically, they are some alphanumeric string S
- The first step of a hash function is to map each key to a big integer
- Let α be the size of the alphabet on which S is written
- Let $char(c)$ be a function that maps each symbol of the alphabet to a unique integer from 0 to $\alpha - 1$

Hashing

- The hash function

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

maps each string to a unique (but large) integer by treating the characters of the string as "digits" in a base-α number system

- The result is unique identified numbers, but they are so large they will quickly exceed the number of slots m in the hash table

- We reduce this number to an integer between 0 and $m - 1$ by taking the remainder of $H(S) \bmod m$

Hashing

- If m, the size of the hash table is selected well, the resulting hash value will be fairly uniformly distributed

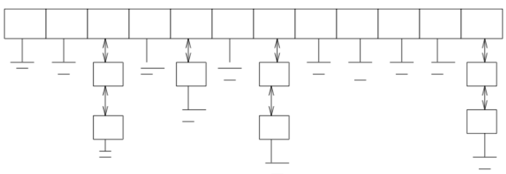
- Ideally, is a large prime not too close to $2^i - 1$

Hashing

- Collisions
  - No matter how good the hash function is, there will sometimes be collisions: two keys mapping to the same number/index
  - One approach to collision resolution: open addressing
    - On insertion, check to see if the desired position is empty
    - If so, insert it
    - If not, find some other place …
    - Simplest approach is sequential probing: look for the next open spot in the table

Hashing

- Collisions
  - No matter how good the hash function is, there will sometimes be collisions: two keys mapping to the same number/index
  - Alternative approach: chaining



Hashing

- Complexity of operations in a hash table
  - Assuming chaining with doubly-linked lists
  - m-element hash table
  - n keys

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

Hashing

- A hash table is often the best data structure to maintain a dictionary
- Also useful for other applications
  - Efficient string matching via hashing

    Problem: given a text string t and a pattern string p,
    does t contain the pattern p as a substring, and if so where?

Hashing

- A hash table is often the best data structure to maintain a dictionary
- Also useful for other applications
  - String matching

    Simplest algorithm:

    - Overlay p in t at every position in the text
    - Check whether every pattern character matches the text character
    - Complexity $O(nm)$, where $n = |t|$ and $m = |p|$

Hashing

- String matching

  Rabin-Karp algorithm:

  - Basic idea: if two strings are identical, so are their hash values
  - If the two strings are different, the hash values are *almost certainly* different (may need to check, but not often)
  - With some clever computation of the hash function (to reduce complexity to constant) the algorithm will usually run in $O(n + m)$

Hashing

- Related applications
  - Is a given document different from all the rest in a large corpus?
  - Is part of this document plagiarized from a document in a large corpus?

