COMP20230: Data Structures & Algorithms Lecture 11: Stack ADTs, Family of Arrays, Doubly Linked Lists and Hash Tables

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Outline

Let's start with some...

Stacks and Hash Tables

Stack ADT Like a queue but single ended.

Hash Tables: Searchable Data Structures

Family of Arrays and Linked Lists

Arrays: (Circular Arrays) and Dynamic Arrays

Linked Lists: Doubly Linked Lists

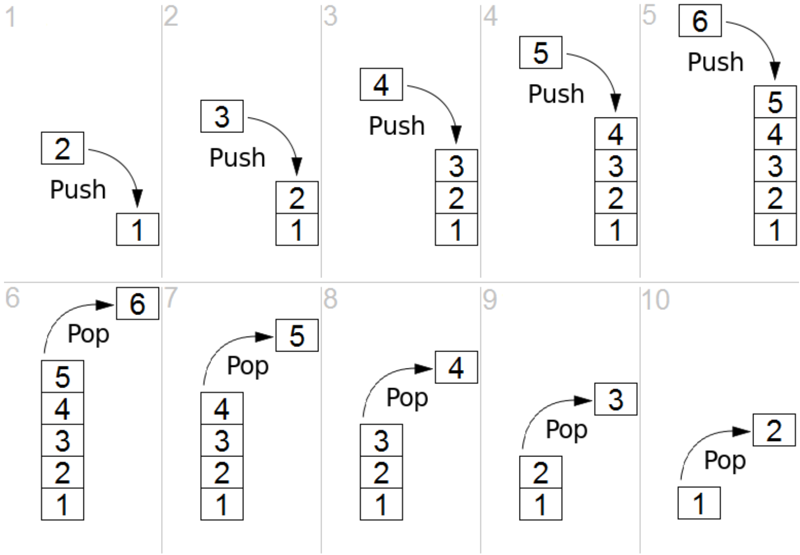
Stack ADT

The stack ADT

ADT for data where elements are piled on each other: only the top element is accessible and new elements are always put on the top of the stack. Think pancakes or a stack of playing cards.

- A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "pushing" onto the stack and "Popping" off the stack removes/retrieves the last item added.

Stack Example



src: `https://upload.wikimedia.org/wikipedia/commons/b/b4/Lifo_stack.png`

Stack: Last In First Out (LIFO)

The stack ADT supports two main methods:

`push(o)`: Inserts object o onto top of stack

Input: Object
Output: none

`pop()`: Removes the top object of stack and returns it; if stack is empty an error occurs

Input: none
Output: Object

Stack ADT

The following support methods should also be defined:

`size()`: Returns the number of objects in stack

Input: none
Output: integer

`is_empty()`: Return a boolean indicating if stack is empty.

Input: none
Output: boolean

`top()`: Return the top object of the stack, without removing it; if the stack is empty an error occurs.

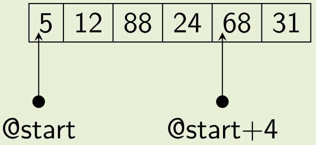
Input: none
Output: Object

Array-based Data Structures

The main characteristic of an array is that the elements can be accessed using an index

- Index can be computed very efficiently: access
- Modification of an element is also very efficient
- **BUT** the modification of the array is complex (sometimes impossible) including editing, adding or deleting elements

An array of size 6.

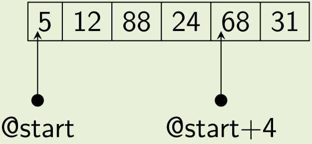


68 is stored in array[4] at memory address start+4

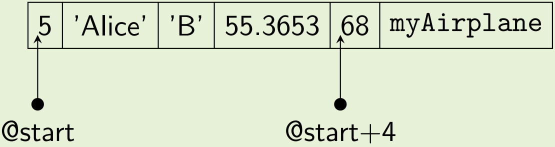
Arrays

What happens if we want to store different sized elements?

From an array of integers of fixed length (e.g. `int16`):

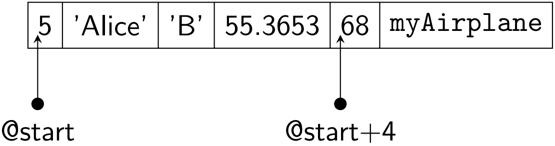


To an array of mixed objects and primitive data types:



Dynamic Arrays

In Python we use a list which is a dynamic array:



Python Lists

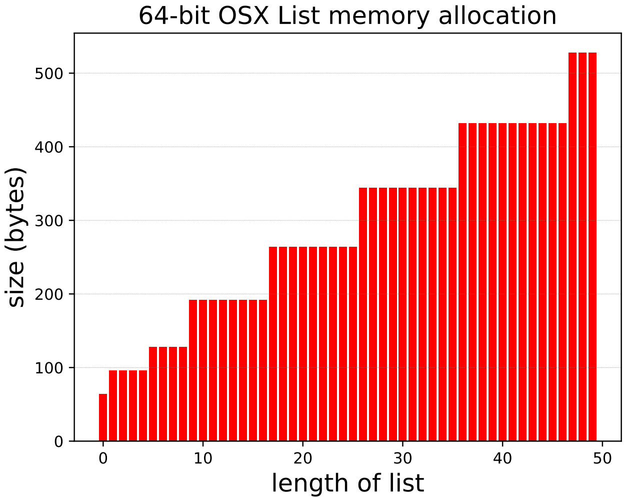
List is Dynamic: Append method

```
import sys
import matplotlib.pyplot as plt
my_list = [] #This is my list
my_list_size =[] #This list will store the size

for i in range(50):
    a = len(my_list)+1
    b = sys.getsizeof(my_list)
    print("Length: ", a, "; Size in bytes: ", b)
    my_list.append(i)
    my_list_size.append(b)

fig, ax = plt.subplots()
plt.bar(my_list,my_list_size, color='r')

ax.grid(color='gray', linestyle=':', linewidth=.2, axis='y' )
ax.set_title('64-bit OSX list memory allocation',fontsize=16)
ax.set_xlabel('length of list',fontsize=16)
ax.set_ylabel('size (bytes)',fontsize=16)
plt.savefig('listSizeFig.pdf')
```

Python Lists

```python
my_list = [] #This is my list
my_list_size =[] #This list will store the size

for i in range(50):
    a = len(my_list)+1
    b = sys.getsizeof(my_list)
    print("Len: ", a,"; Size in bytes: ",b)
    my_list.append(i)
    my_list_size.append(b)
```

List Memory Growth

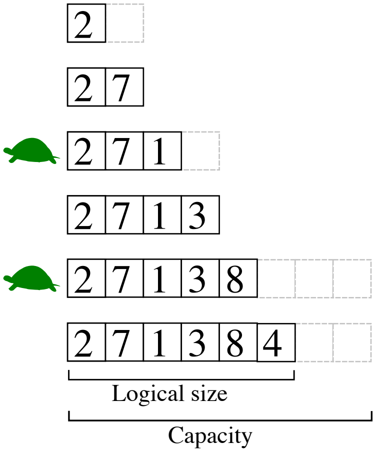


Dynamic Arrays

The array is created with more capacity than it needs, i.e. real capacity > logical size.

The turtles highlight where the slow operations are. $\boxed{2}\boxed{7}$ has used the real capacity so to add the $\boxed{1}$ we need to resize the array (and may even need to move it in memory

Adding in the middle is also a challenge



src: https://upload.wikimedia.org/wikipedia/commons/3/31/Dynamic_array.svg

Dynamic Array: Insert at end

Algorithm **insert_at_the_end**

Input: DA a dynamic array, s and c two integers representing the size and the capacity of DA, e an element
Output: the size of DA grows by 1 and e is inserted at the end of DA

```
if s = c then
    increase the capacity by a factor of X (you can pick whatever you think if the best progression here)
    For instance:
    Increase the capacity to c ← c × 2
end if
DA[s] ← e
s ← s + 1
```

Dynamic Array: Insert (not end)

Four in the bed and the little one said... roll over (but don't fall out!)

The difference here is that we need to shift all the subsequent elements up an index in the array

---

Algorithm **insert_not_at_the_end**

Input: DA a dynamic array, s and c two integers representing the size and the capacity of DA, e an element that we wish to insert at rank i

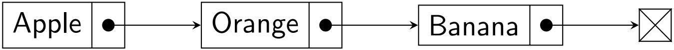
Output: the size of DA grows by 1 and e is inserted at position i

```
if s = c then
    increase the capacity by a factor of X (you can pick whatever you think if the best progression here)
    For instance:
    Increase the capacity to c ← c × 2
end if
for j = i to s do
    DA[j + 1] ← DA[j]
end for
DA[i] ← e
s ← s + 1
```

Doubly Linked Lists

Recall: Linked Lists are

a set of *element bearing nodes* **threaded together**



Example: Doubly Linked List

Two links out of each node.

Doubly Linked Lists

Nodes can have more than one pointer

e.g. doubly linked lists have nodes with two pointers
Advantage: Operations are simpler (no need to keep track of current + previous + next)
Disadvantage: Uses more memory

Doubly Linked Lists

Nodes can have more than one pointer

e.g. doubly linked lists have nodes with two pointers
Advantage: Operations are simpler (no need to keep track of current + previous + next)
Disadvantage: Uses more memory

Could we have a triple linked list?

What might it be useful for?

Python List

The python list is like a swiss army knife

General utility class that can be applied to many problems and situations

Python list as a set, sequence, stack or queue

`https://docs.python.org/3/tutorial/datastructures.html`

Let's take a look...

Searching data structures

Example Symbol Tables

Application	Purpose of Search	Key	Value
dictionary	find word definition	word	definition
book index	find relevant pages, word occurrences	term	list of page numbers
account management	process transaction	account number	transaction details
web search	find relevant web pages	keyword	list of page titles and urls
compiler	find type and value of variable	variable name	type and value

ADT of a symbol table

For an **unordered symbol table** the ADT has the following operations:

`put(key, value)`	put key-value pair into the table
`get(key)`	value paired with key (null if key is absent)
`delete(key)`	remove key from table and value paired with key
`contains(key)`	is there a value paired with key?
`isEmpty()`	is the table empty?
`size()`	number of key-value pairs in the table
`keys()`	all the keys in the table

ADT of a symbol table

For an **unordered symbol table** the ADT has the following operations:

`put(key, value)`	put key-value pair into the table
`get(key)`	value paired with key (null if key is absent)
`delete(key)`	remove key from table and value paired with key
`contains(key)`	is there a value paired with key?
`isEmpty()`	is the table empty?
`size()`	number of key-value pairs in the table
`keys()`	all the keys in the table

Aside: Ordered Symbol Table ADT

If we want to keep our symbols ordered, we need to keep information about their rank and a number of other operations are required: `min()`, `max()`, `floor(key)`, `ceiling(key)`, `rank(key)`, `select(rank)`, `deleteMin()`, `deleteMax()`, `size(low_key,high_key)`, `keys(low_key,high_key)`

Searching data structures

Three classic data structures that can support efficient searchable symbol-table implementations:

1. Hash tables
2. Binary search trees
3. Balanced search Trees: 2–3 Trees, Red-black trees, AVL Trees

Hash figures adapted from:

Algorithms (Sedgewick & Wayne)

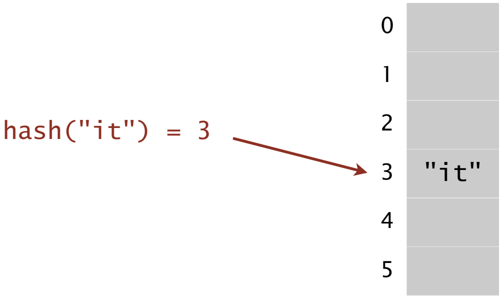
Hash Tables

Hash Tables

Save items in a key-indexed table (index is a function of the key)

Hash Function

Method for computing array index from a key.



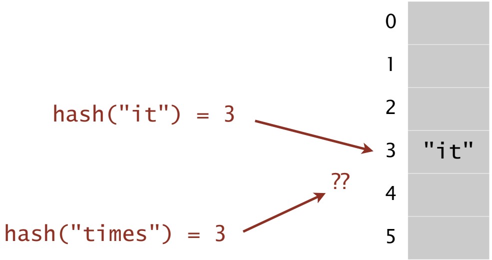
Hash Tables: Requirements and Issues

Compute the hash function

Good algorithm (i.e. fast, efficient, scalable etc.)

Collision resolution

Algorithm and data structure to handle two keys that hash to the same array index



Example: Python Dictionary

```python
airports={"JFK": ("John F Kennedy Intl","United States",40.639751, -73.778925),
          "SYD": ("Sydney Intl","Australia",-33.946111,151.177222),
          "LHR": ("London Heathrow","United Kingdom",51.4775,-0.461389)}

# print a search result
print(airports["SYD"])
print("Airport Keys: ", airports.keys())

# add an airport to the dictionary
airports["AMS"]=("Schiphol","Netherlands",52.308613,4.763889)

# store the value of a search and print it
destination=airports.get("AMS")
print(destination)

# pop (search and remove) a value from dict and save it in a variable
oz_airport = airports.pop("SYD")
print("Airport Keys: ", airports.keys())

# what is the hash for key AMS?
# Does it change if I call it twice? What if I rerun the program?
print("AMS hash is:", hash("AMS"))
print("AMS hash is:", hash("AMS"))
print("DUB hash is:", hash("DUB"))
```

Output:

```
('Sydney Intl', 'Australia', -33.946111, 151.177222)
Airport Keys:  dict_keys(['JFK', 'SYD', 'LHR'])
('Schiphol', 'Netherlands', 52.308613, 4.763889)
Airport Keys:  dict_keys(['JFK', 'LHR', 'AMS'])
AMS hash is: 6708379502801481095
AMS hash is: 6708379502801481095
DUB hash is: -3052993293245237079
```

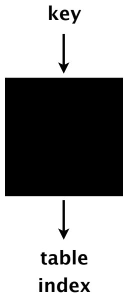
Hash Tables: Computing the Hash Function

Ideally: Scramble the keys uniformly to produce

Equally computable table index
Each table index equally likely for each key.

Hash Codes

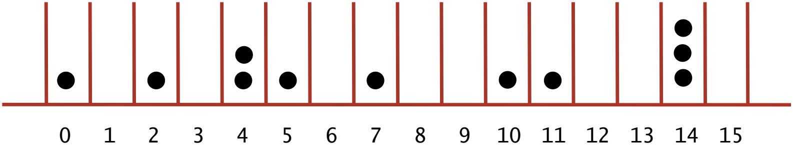
Integers, e.g.
Most significant part of a float;
Memory address of an object



Hash Tables

Uniform Hashing Assumption

Each key is equally likely to hash to an integer between 0 and $M - 1$.

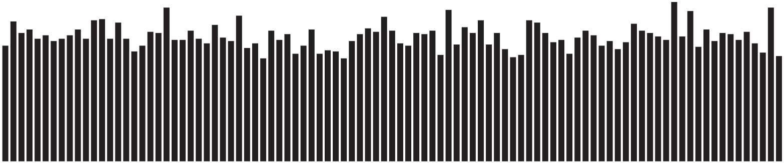


Bins and Balls

Evenly distribute balls into the slots of a hash table.
Throw balls aiming for uniform distribution at M bins.

Example Hash Table

Java hash table implementation result for distributing keys of strings (words) in Tale of Two Cities. (M=97)

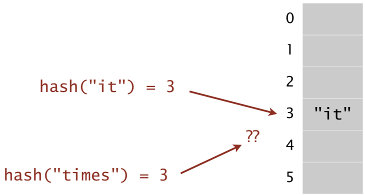


Hash value frequencies for words in Tale of Two Cities (M = 97)

Hash Tables

Collisions

Two distinct keys hashing to same index
Collisions inevitable (unless *dynamic perfect hashing* implemented – memory hungry!).



Birthday Problem

How many birthdays on the same day in a class of 70? With only 23 people, the probability that two people have same birthday is 50%

Hash Tables

Implementation

Separate Chaining Symbol Table
Linear Probing

Separate Chaining Symbol Table

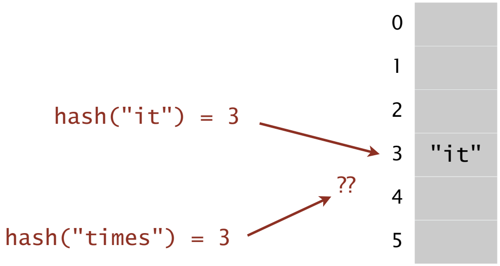
M lists and N keys.

Use an array of $M < N$ linked lists

`Hash`: Map key to integer i between 0 and $M - 1$

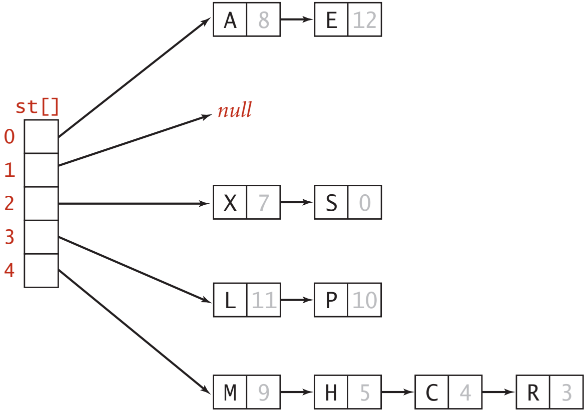
`Insert`: Put at front of ith chain (if not already there)

`Search`: Need to search only ith chain



Separate Chaining Symbol Table

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



Separate Chaining Symbol Table

Getting the balance right: what size for balance between `insert` and `search`?

Analysis

Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N/M is extremely close to 1

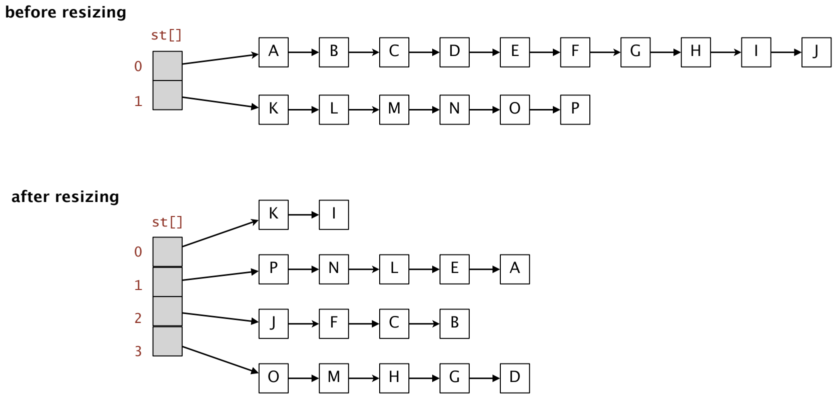
Consequences

Number of probes for `search`/`insert` is proportional to N/M
M too large $\Rightarrow$ too many empty chains
M too small $\Rightarrow$ chains too long
Typical choice: $M \sim N/4 \Rightarrow$ constant-time ops

Separate Chaining Symbol Table

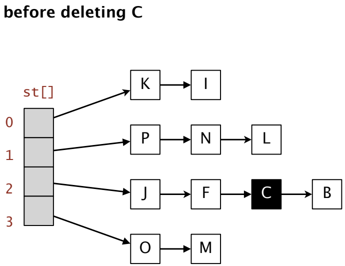
Resizing: Average length of list $N/M = constant$

Double size of array M when $N/M \geq 8$
Halve size of array M when $N/M \leq 2$
Need to rehash all keys when resizing



Separate Chaining Symbol Table

Deleting is straight-forward



Collision Resolution Strategy: Use Open Addressing

Open addressing

When a new key collides, find next empty slot, and put it there

st[0]	jocularly
st[1]	*null*
st[2]	listen
st[3]	suburban
⋮	*null*
st[30000]	browsing

Linear-probing Hash Table

Linear-probing

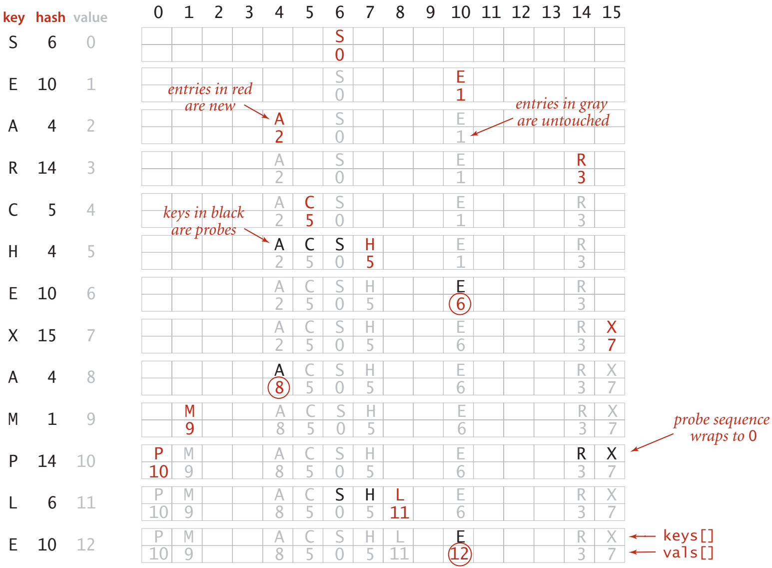
Open addressing scheme for resolving collisions in hash tables

`Hash:` Map key to integer i between 0 and $M-1$ `Insert:` Put at table index i if free; if not try $i+1$, $i+2$, etc. `Search:` Search table index i; if occupied but no match, try $i+1$, $i+2$, etc.

Note

Array size M must be greater than number of key-value pairs N

Example of Linear Probing (video on moodle)



Linear Probing Hash Table

Resizing: Average length of list $N/M \leq 1/2$

Double size of array M when $N/M \leq 1/2$
Halve size of array M when $N/M \geq 1/8$
Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

Linear Probing Hash Table

Deletion: What happens if we delete S from hash table?

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if hash(H) = 4

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

Linear Probing Hash Table

Deletion: What happens if we delete S from hash table?

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

after deleting S ?

doesn't work, e.g., if hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

Cannot just leave null/None – will not find H

Need to rehash the cluster to the right of the deleted key.