CS2521: Data Structures

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Where are we?

- We know how to specify an algorithm in pseudocode.
- We can show that an algorithm is correct.
- We are able to compare algorithms in terms of efficiency.

So now what?

- Algorithms store and/or manipulate data.
- Such data is stored in several commonly used structures, e.g.
  - arrays
  - lists
  - dictionaries (a.k.a maps, hashes)
- Such structures exemplify classes of abstract data types.
- Many different implementations of each such data structure can be found.
- Changing the implementation does not affect algorithm correctness
- But such changes can affect the time taken to execute different operations.
- We will be examining the properties of different implementations of various abstract data structures.

Contiguous & Linked Data structures

- One way to classify data structures is to split them by whether they are contiguous or linked.
  - A contiguous data structure occupies a single chunk of memory.
  - A linked data structure is made up of distinct chunks of memory.
- Contiguous:
  - No need for elements of the data structure to reference other parts of the structure.
  - Arrays, heaps, hash tables
- Linked:
  - Elements of the data structure hold references to other elements
  - Lists, trees
- Each has advantages and disadvantages.

Arrays

- Probably the most fundamental data structure.
- Consists of fixed-size data records, allowing efficient location of element by index or address.
- E.g. If we know that the first element of the array is at position 0x5555 in memory, and that each element is 4 bytes long, the 5th element will be in position 0x5569.
- Computers can typically retrieve any element of memory by its address in constant time.
- Advantages:
  - Constant time access by index
  - Space efficient
  - Physically continuous in memory, allowing fast cache access to later elements
- Disdavantages:
  - No ability to alter size dynamically.

Overcoming array size limitations

```
1: function addElement(e,array)
2:     create a new array na of size |array|+1
3:     for all i=0 to |array|-i do
4:         na[i]=array[i]
5:     end for
6:     na[|array|]=e return na
7: end function
```

- Adding n elements to the array requires $n-1$ copies for first element, $n-2$ copies for second, etc. Complexity?

Overcoming array size limitations

```
1: function addElement(e,array)
2:     create a new array na of size |array|+1
3:     for all i=0 to |array|-i do
4:         na[i]=array[i]
5:     end for
6:     na[|array|]=e return na
7: end function
```

- Adding n elements to the array requires $n-1$ copies for first element, $n-2$ copies for second, etc. Complexity? $\Theta(n^2)$.
- Or we could allocate a very large array initially (wastes memory)

A better approach

```
1: function addElement(e,array)
2:     if array is full then
3:         create a new array na of size 2*|array|
4:         for all i=0 to |array|-i do na[i]=array[i]
5:     end if
6:     na[|array|]=e
7:     return na
8: end function
```

- Now adding n elements requires $\log n$ resizes of the array.
- Total number of times elements are moved (**Homework:** show this, discussed in practical):

$$\sum_{i=1}^{\log n} in/2^i = n\sum_{i=1}^{\log n} i/2^i \leq n\sum_{i=1}^{\infty} i/2^i = 2n$$

- amortised analysis: total work as $n \to \infty$ is the same as having allocated the correctly sized array at the start!

Pointers and Linked Structures

- Pointers represent the in memory address of a data structure.
- Pointers are used to hold linked structures together.
- Basic structure: Linked List
  - Each node holds a value and a pointer to the next node.
  - Basic operations: search, insert, delete
  - Double linked lists have a reference to the previous node. Requires more space, but simplifies some operations.
  - The head of the list is its first node; a reference to it is always needed.

List Search and Insertion

```
1: function search(node,value)
2:     if node.value==value then return true
3:     else if node.next==null then return false
4:     return search(node.next,value)
5: end function
6: function insert(node,value,index)
7:     if index==0 then
8:         n=new Node
9:         n.value=value; n.next=node.next
10:        node.next=n
11:        return
12:    end if
13:    insert(node.next,value,index-1)
14: end function
```

List Deletion

```
1: function delete(node,count)
2:     if count=1 then
3:         node.next=node.next.next
4:         delete node.next from memory as needed (via temporary variable)
5:         return
6:     else
7:         delete(node.next,count-1)
8:     end if
9: end function
```

- We used recursion, iterative approach also simple.
- Linked List advantages
  - No overflow
  - Simple insertion and delete
- Disadvantages
  - Extra space for pointers
  - No random access
  - Poor cache performance

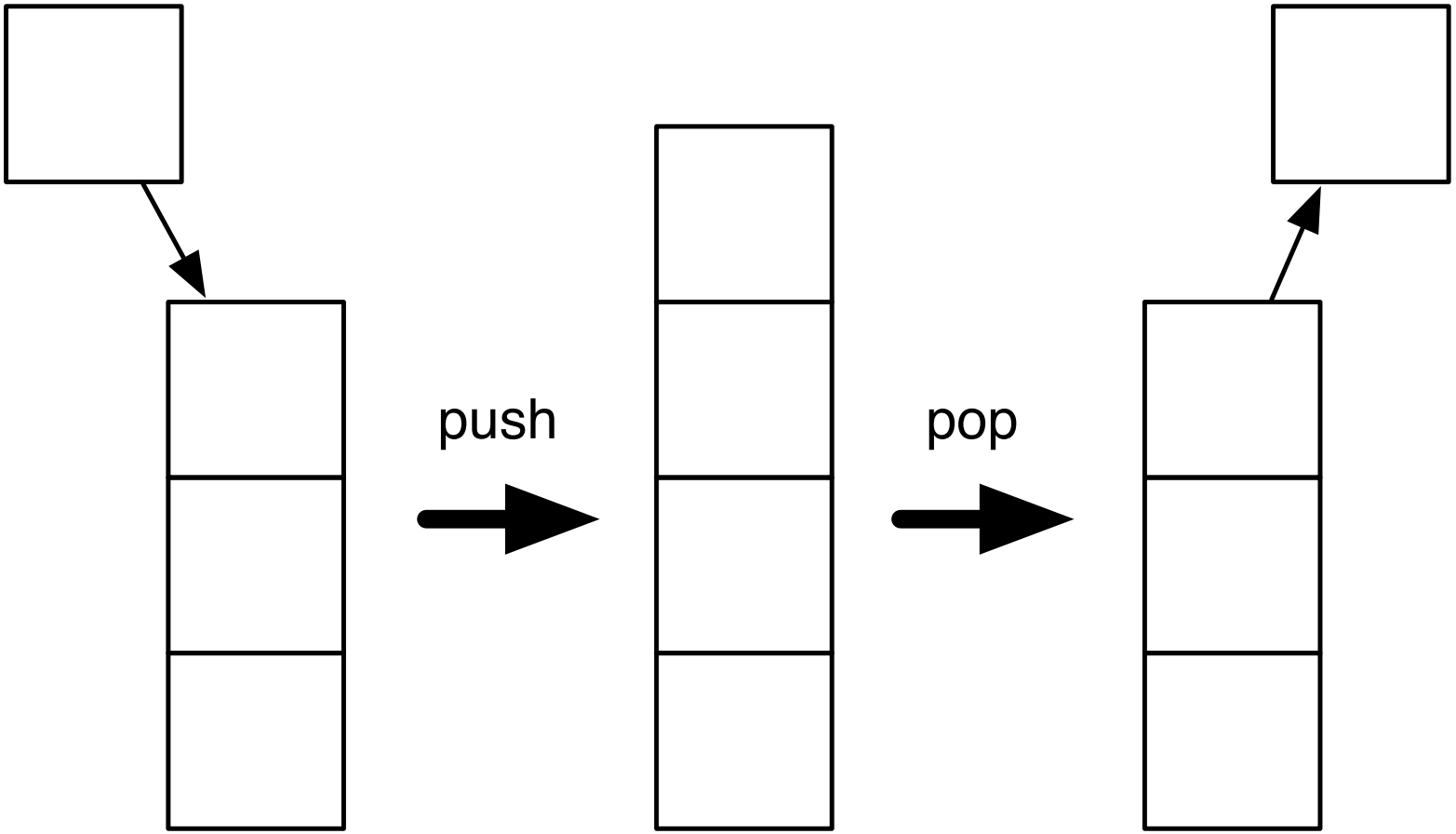
Recursive Structures

- Note that both arrays and lists are recursive structures.
- Removing the first element of a list leaves a (smaller) list.
- Removing the nth element of an array leaves 2 smaller arrays.

Stacks and Queues

- Stacks and queues are containers that can allow data retrieval independent of the contents of the container (unlike lists and arrays, where an index is required).
- Both support operations for adding, and removing items from the container.
- Stacks are LIFO (last in first out) objects.
- Queues are FIFO (first in first out) objects.

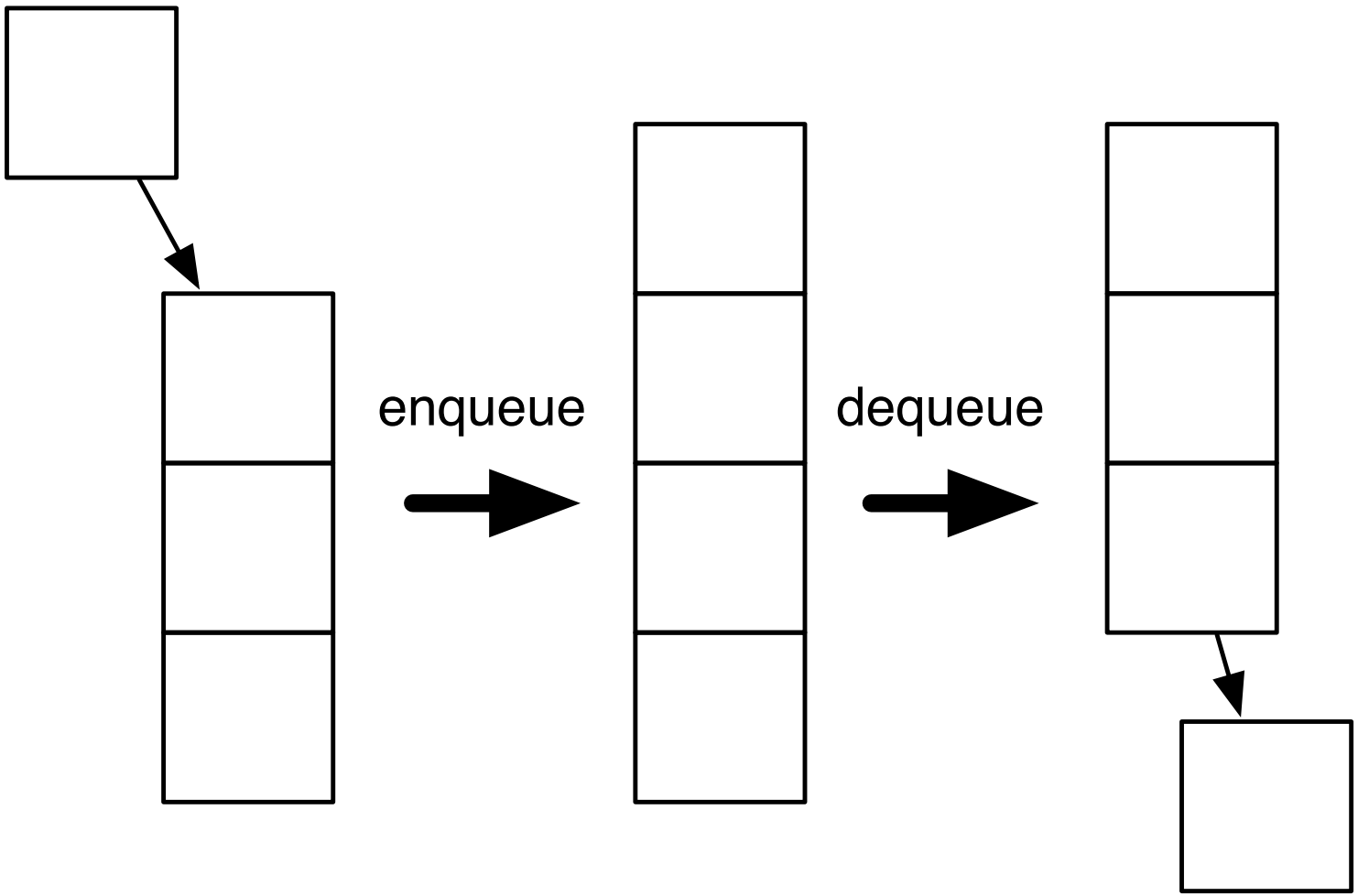
Stack



Stack

First object in is last object out

Queue



First object in is first object out

Stacks and Queues

- Stacks are very efficient, used in many real world situations (e.g. the java call stack)
- Queues are used in scheduling as it's typically "fair" that the first job is dealt with first. Queues minimise maximum waiting time before being seen.
- Average time until processing is the same for both structures.
- Both can be efficiently implemented using arrays (if the maximum number of items in the container is known ahead of time) or linked lists (if it is not).

Dictionaries

- A dictionary, map or association links data to a key.
- Basic operations:
  - *search(D,k)*: Given a key k returns the item associated with the key.
  - *insert(D,k,x)*: adds the data item x to dictionary D, associating it with key k.
  - *delete(D,k)*: Given a key k, remove the data item (and key) from the dictionary.
- Different underlying dictionary implementations allow you to efficiently identify the item associated with the largest/smallest key, or retrieve items in the order of the keys.
- Dictionary Examples: phonebook, name is the key, phone number is the data. Marks: studentid is the key, the list of marks is the data.

Implementing Dictionaries

- Create an array of (*key*, *value*) pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys.

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index,

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1.

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1. $\Theta(1)$
- To identify max/min/predecessor/successor

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1. $\Theta(1)$
- To identify max/min/predecessor/successor we have to iterate through the array. $\Theta(n)$
- If the array is sorted, then
  - Search:

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1. $\Theta(1)$
- To identify max/min/predecessor/successor we have to iterate through the array. $\Theta(n)$
- If the array is sorted, then
  - Search: $\Theta(\log n)$
  - Insert,delete:

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1. $\Theta(1)$
- To identify max/min/predecessor/successor we have to iterate through the array. $\Theta(n)$
- If the array is sorted, then
  - Search: $\Theta(\log n)$
  - Insert,delete: $\Theta(n)$
  - max/min/predecessor/successor:

Implementing Dictionaries

- Create an array of $(key, value)$ pairs, store n, the maximum index.
- To search, iterate through the array, comparing keys. $\Theta(n)$
- To insert, increment n and add the $(key, value)$ at the next index, $\Theta(1)$
- To delete, write the last element of the array to the deleted index, reduce n by 1. $\Theta(1)$
- To identify max/min/predecessor/successor we have to iterate through the array. $\Theta(n)$
- If the array is sorted, then
  - Search: $\Theta(\log n)$
  - Insert,delete: $\Theta(n)$
  - max/min/predecessor/successor: $\Theta(1)$
- The underlying representation should be tuned to the envisioned use of the dictionary.
- **Homework:** Perform the same analysis for sorted and unsorted single and doubly linked lists. To be discussed in practical.

Trees

- A rooted binary tree is either
  1. Empty
  2. a node (called the root) and two binary search trees (called left and right).
- A binary search tree associates a key with each node such that

  $$\text{left.key} < \text{root.key} < \text{right.key}$$

- For any set of keys, there is only one legal binary search tree.
- Nodes typically also store a data value.
- BSTs typically support search, insertion, deletion and traversal

Search, Minimum and Maximum

```
1:  function search(node,key)
2:      if node is empty then
3:          return null
4:      else if node.key==key then
5:          return node.value
6:      else if key<node.key then
7:          return search(node.left,key)
8:      else
9:          return search(node.right,key)
10:     end if
11: end function
```

```
1: function min(node)
2:     if node.left is empty then
3:         return node.value
4:     else
5:         return min(node.left)
6:     end if
7: end function
```

Maximum uses right instead of left.

Traversal and insertion

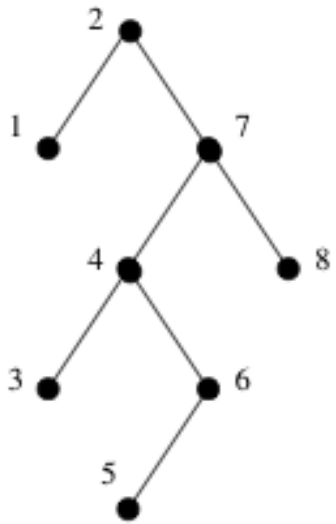
```
1: function traverse(node)
2:     if node is empty return
3:     traverse(node.left)
4:     doSomething(node.value)
5:     traverse(node.right)
6: end function
```

- Traversal performs an operation on every node on the tree
- To insert a node (**Homework**: write pseudocode for this, discussed in practical)
  1. Search the tree until you get to an empty node.
  2. Replace the empty node with a new node containing the key and value.

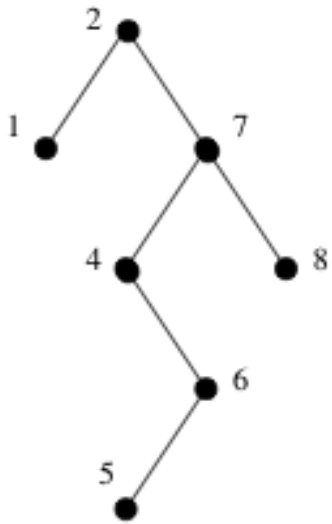
Deletion

- Deletion of a leaf node is easy, just set it to empty.
- Deletion of a node with 1 child is easy, replace it with its child.
- Deletion of a node with 2 children is harder: replace with the key of its immediate successor (or immediate predecessor). To achieve, find minimum node from right node of the node to be deleted and move this node to the deleted node's position.

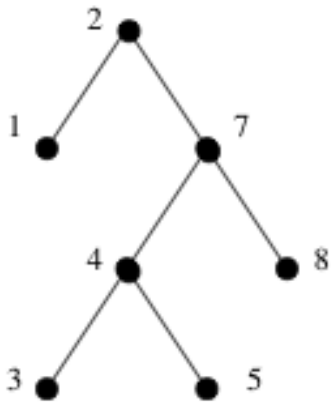
Deletion



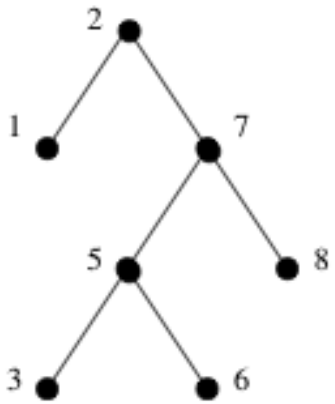
initial tree



delete node with zero children (3)



delete node with 1 child (6)



delete node with 2 children (4)

Costs

- Traversal visits all nodes of the tree, i..e $\Theta(n)$
- All other operations depend on the height of the tree, so $\Theta(h)$. This height can be anything from n to $\log n$.
- A balanced search tree ensures that the height of the tree is always $\log n$.

Priority Queues

- A priority queue stores a list of keys and values, and allows one to
  1. Insert a new key/value pair into the queue
  2. Find maximum (or find minimum) which returns a pointer to the item with the maximal (or minimal) key value
  3. Delete maximum (or delete minimum) which removes the element with the smallest associated key from the list.
- How can we implement a priority queue?
  - Unsorted array
  - Sorted array
  - Balanced binary search tree

Complexity

	Unsorted Array	Sorted Array	Balanced Tree
Insert	O(1)	O(n)	O(log(n))
Find-Min	O(1)	O(1)	O(1)
Delete-Min	O(n)	O(1)	O(log(n))

- Delete-minimum: find-minimum then delete.
- But how is find-minimum O(1) for unsorted array?

Complexity

	Unsorted Array	Sorted Array	Balanced Tree
Insert	O(1)	O(n)	O(log(n))
Find-Min	O(1)	O(1)	O(1)
Delete-Min	O(n)	O(1)	O(log(n))

- Delete-minimum: find-minimum then delete.
- But how is find-minimum O(1) for unsorted array?
- Keep a pointer to the minimal element
- How is delete-minimum O(1) for sorted array?

Complexity

	Unsorted Array	Sorted Array	Balanced Tree
Insert	O(1)	O(n)	O(log(n))
Find-Min	O(1)	O(1)	O(1)
Delete-Min	O(n)	O(1)	O(log(n))

- Delete-minimum: find-minimum then delete.
- But how is find-minimum O(1) for unsorted array?
- Keep a pointer to the minimal element
- How is delete-minimum O(1) for sorted array?
- Reverse sort array, delete last element and decrement the array size.

Hashing and Strings

- A hash function maps a key to an integer.
- We can then store an object in an array at the index obtained from a hash function.
- How can we generate a hash function?
- Let's assume that our key is a string S derived from some alphabet.

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

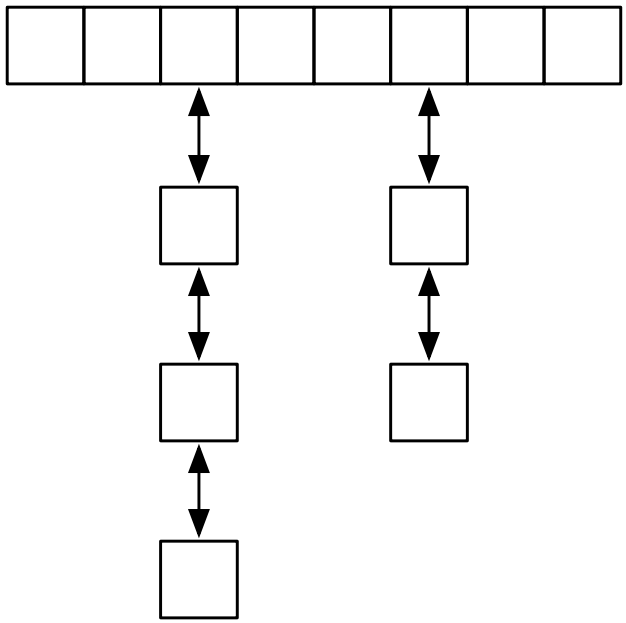
- Here $char(s_i)$ maps a single element from the string to an integer between 0 and $\alpha - 1$
- E.g. $\alpha = 10$, a=1,b=2,c=3. "aab"=112
- But our array would have to be huge to cope with large strings.
- If we have an array of size m, we need to reduce $H(S)$ to between 0 and $m - 1$.
- Simple approach: divide $H(S)$ by m and take the remainder: $H(S)$ mod m

Collisions

- Different keys will occasionally hash to the same value. This is known as a collision.
- To deal with collisions, we can use
  - Chaining
  - Open addressing
- Hash tables are often the best structures for storing dictionaries.

Chaining

- We represent the hash table as an array of buckets.
- Each bucket is a linked list.
- The i'th bucket holds all items that hash to the value i.
- If n keys are in a table with m buckets, and the distribution is approximately uniform, each bucket will hold n/m items. Complexity of search?

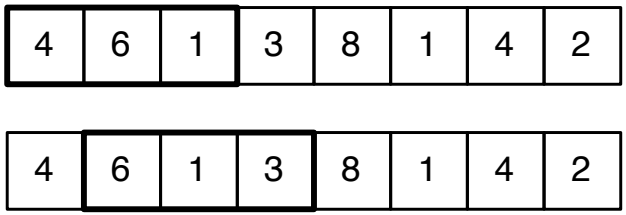


Open Addressing

- Here, the hash table is an array of elements, all initialised to some empty value.
- To insert, we check if the position is empty, if so, insert. If not we need to find another place to insert it.
- Sequential probing: insert the item in the next open spot in the table. Works well if the table is not too full.
- Any other suggestions for probing?
- Search requires going to the table, checking for the item, if not found, going to the next spot etc, until item is found or empty is detected.
- Delete requires reinserting all items in the run following the deleted element (very ugly).
- Open addressing does not require us to store (lots of) pointers.
- What are traversal complexities for chaining and open addressing?

So what else can we do with hashes?

- String matching was $O(nm)$. n is document length, m is target length.
- Can we do better?
- We have a sliding window of length m over the document, requiring $n - m + 1$ window positions, each taking m comparisons.

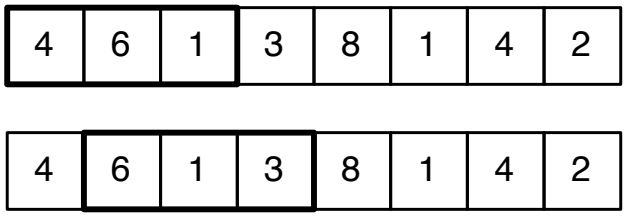


So what else can we do with hashes?

- Hash the target string. Hash the sliding window. e.g.
- Let $\alpha = 3$.

$$H(S, j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

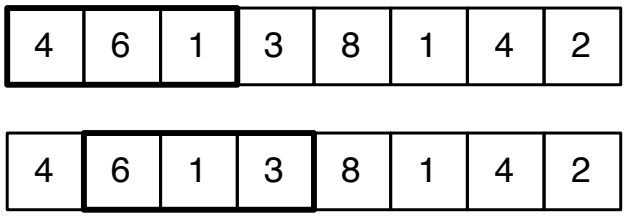
- $3^2 \times 4 + 3^1 \times 6 + 3^0 \times 1 = 55$
- $3^2 \times 6 + 3^1 \times 1 + 3^0 \times 3 = 60$



So what else can we do with hashes?

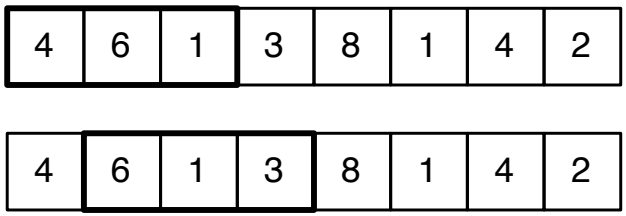
$$H(S, j+1) = \alpha(H(S, j) - \alpha^{m-1} \times char(s_j)) + s_{j+m}$$

- We can calculate this in constant time!
- in case of a hash match, check the entire string ($O(m)$)



So what else can we do with hashes?

- This still works if we compute $H(S, j)$ mod M. (keeps hash small)
- This is essentially the Rabin-Karp algorithm for string matching.
- This algorithm can run in $O(n + m)$ if no collisions occur, but in the worst case we have nearly constant collisions, i.e. $O(nm)$.
- If $M \approx n$, 1 false collision per string, if $M \approx n^k$ typically no collisions.



What else can we do with hashes?

- Duplicate detection (e.g. plagarism software)
- Ensure file hasn't changed
- Provide secrecy guarantees
  - Consider a secret auction with no adjudicator. How can we keep bids secret while preserving honesty?

Summary

- We've examined various data structures:
  - Arrays, lists
  - Stacks, queues
  - Dictionaries (implemented using arrays/lists and hashes)
  - Trees
  - Priority queues
- Each has their place, depending on exact application; different tradeoffs in complexity for different operations, there is no "best data structure".