Lecture 2 – Elementary Data Structure

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Contact

- My email: yings@hkust-gz.edu.cn
  TA: Hongbo Yin
  (hyin744@connect.hkust-gz.edu.cn)

Outline

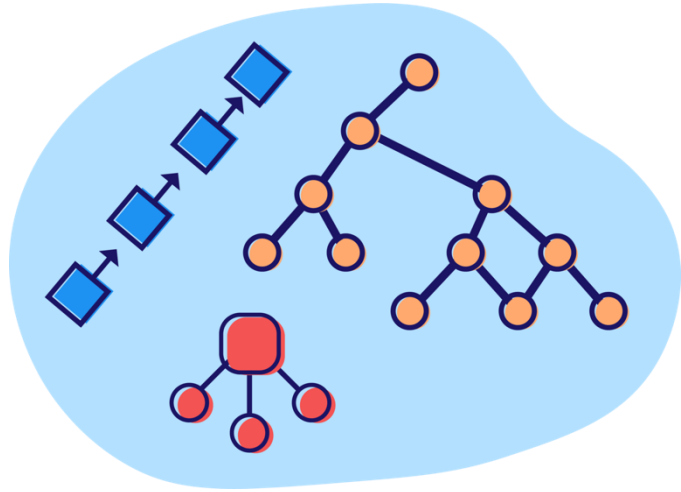
- Data Structure
- Array and Linked Lists
- Queues and Stacks
- Graphs and Trees
- Hash Tables

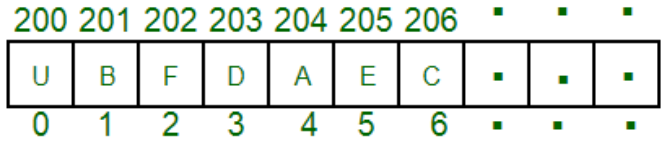
Data

- Dictionary Definition: entities, characters, or symbols that a computer processes through operations
  - Example: In the expression c = a + b, 'a' and 'b' represent data
- Data in itself might not always be meaningful or usable
  - 'NyaIMsYgienm' represents **data**, but lacks clear information
  - 'MyNameIsYing' is both **data** and meaningful **information**
- For data to convey information effectively, it needs to be well-organized and managed

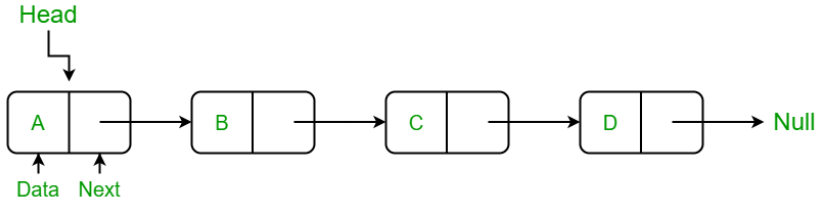
Data Structure

- Data structure: a systematic arrangement of data that facilitates efficient
  - Storage
  - Retrieval
  - Operations
- Key Components
  - Establishing relationships among data elements.
  - Defining potential operations to be applied on data.
- Some common data structures
  - Arrays, Linked Lists, Queues, Stacks, Hash Tables, Trees, Graphs

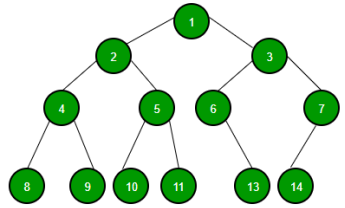




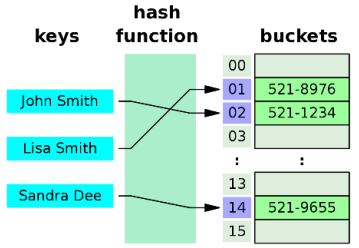
Array



Linked list



Trees



Hash Tables

Arrays and linked lists

Abstract Data Type: Linear List

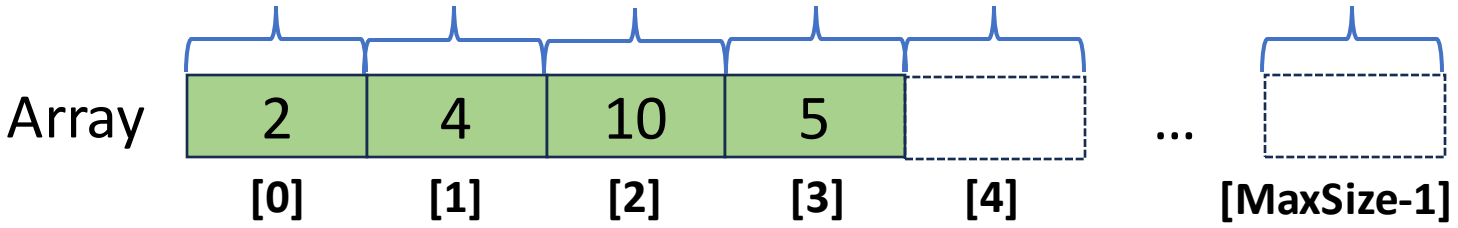
- Ordered sequence formed as $[e_1, e_2, \ldots, e_n]$
- Common operations
  - Access: Retrieve the *k*-th element.
  - Search: Find a specified element.
  - Delete: Remove an element.
  - Insert: Add a new element (at a specified position).
  - Append: Add a new element to the end.
  - Concatenate: Join two lists together.
- Implementations: Array and Linked lists

Array

Simplest way to store a list of elements

- Stores a fix-size sequence of fix-type elements in contiguous memory location
- Intuitive understanding: a box divided into small fix-size rooms

Same room for each element



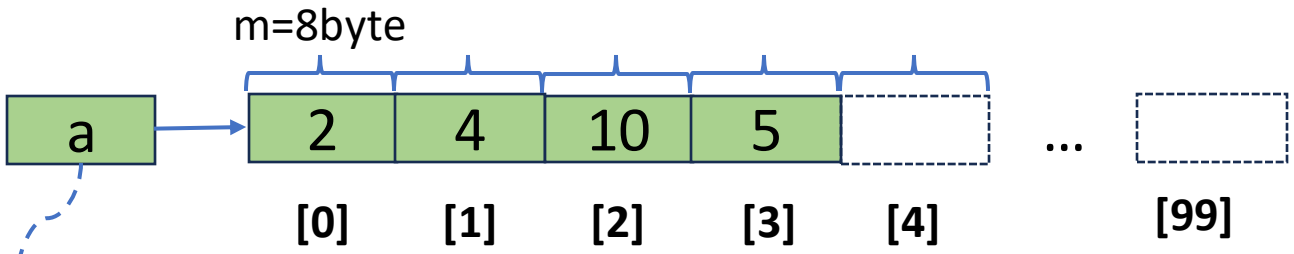
Even not assigning data, the space is pre-allocated

Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element
- Delete an element
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

Example: (C/C++) int a[100];



Stores the address of the first element

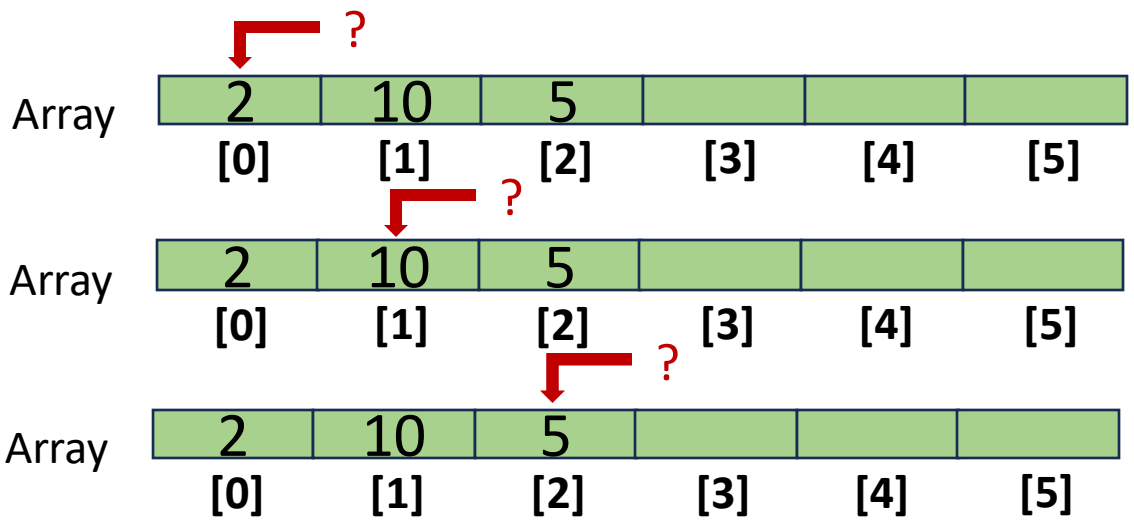
- Access a[k] at a + m * k

Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element: O(n)
- Delete an element
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

```
for i in range(n):
    if a[i] == m:
        return i
```

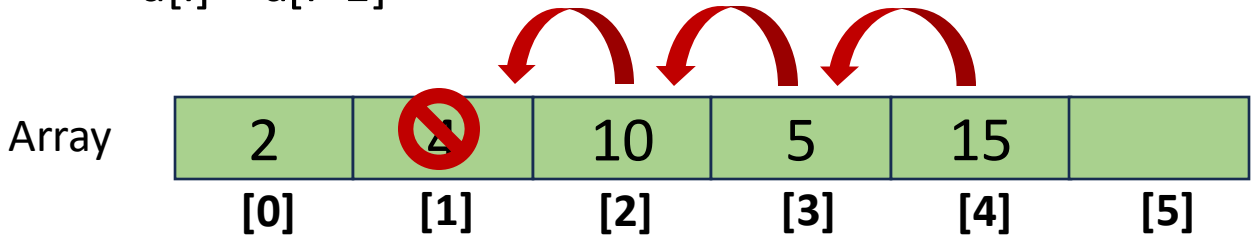


Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element: O(n)
- Delete an element: O(n)
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

```
for i in range(k, n-1):
    a[i] = a[i+1]
```



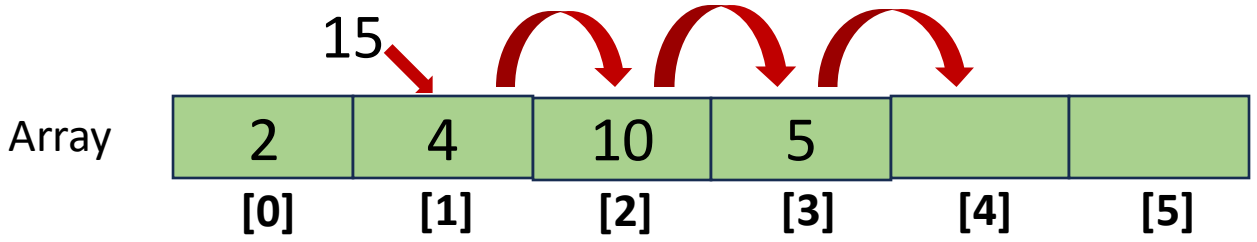
Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element: O(n)
- Delete an element: O(n)
- Insert an element at certain position: O(n)
- Append a new element at the end
- Concatenate two lists

```
for i in range(n-1, k, -1)
    a[i+1] = a[i]
a[k] = m
```

(Supposing the array is not full)



Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element: O(n)
- Delete an element: O(n)
- Insert an element at certain position: O(n)
- Append a new element at the end: O(1)
- Concatenate two lists

l: current length

a[l] = m

l = l + 1

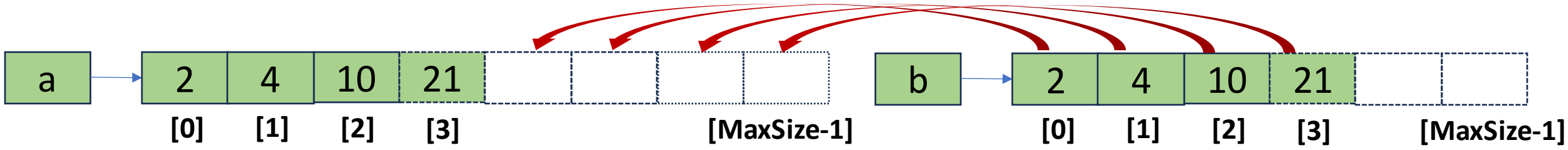
Array

List operations with array

- Access the k-th element: O(1)
- Search for a given element: O(n)
- Delete an element: O(n)
- Insert an element at certain position: O(n)
- Append a new element at the end: O(1)
- Concatenate two lists: O(n)

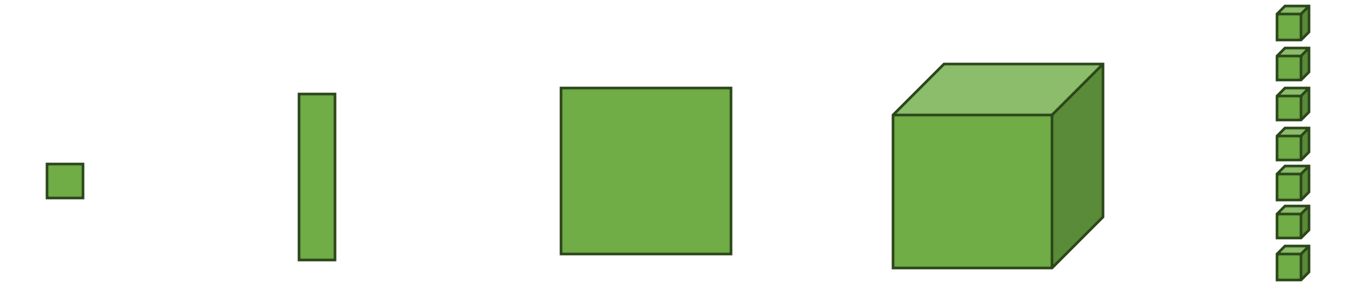
(supposing the allocated length is enough)

```
for i: 0 to m-1:
    a[n + i] = b[i]
```



Multi-Dimensional Array

- n-d array can be implemented with 1-d array



- Example: $N \times M$ array implemented with a 1-d array with length NM

0	1	2	3	4	5	6	…	NM-1

0	1	2	…	M-1
M	M+1	M+2	…	2M-1
…	…	…	…	…
(N-1)M	(N-1)M+1	(N-1)M+2	…	NM-1

Interface: a[i, j]

- Allocate NM-length 1d array
- Regard each M elements as a group
- a[i,j] is the element at $a + iM + j$

Example of Array in Python

- NumPy array

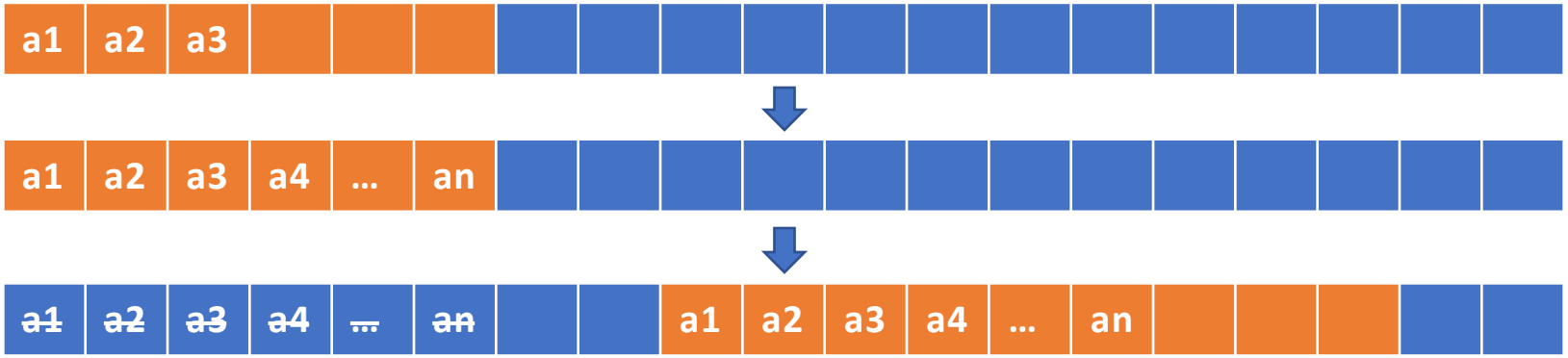
  - How to implement the reshape operation?
  - Answer: simply change N and M (backend still indexing on 1-d array)
- Python built-in list
  - Why unfix-length, mix-typed (different size for each element)

```
lst = [1, 'dog', 1.5, []]
lst.append(666)
lst[0]= ’cat’
```

Python List

Unfix-length

- Pre-allocate the space (not visible to the user)
- Copy to a larger space when use up: O(n)



Python List

Mix-typed (different size)

- Store address instead of value

Each address points to the real element

Summary

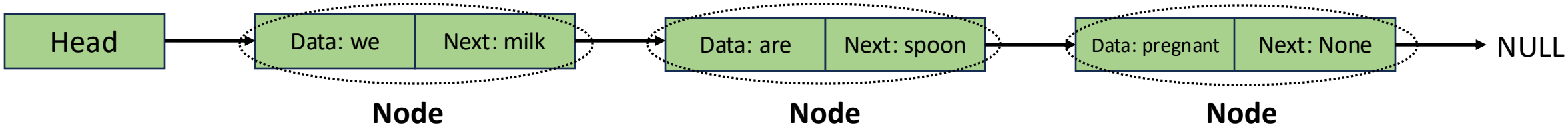
	Array
Pros	• Efficient access • Space efficient
Cons	• Fixed size or requires resizing • Expensive (O(n)) insertions/deletions • Contiguous memory allocation can be an issue

Linked List

- Another linear data structure, allow dynamic space allocation
- The elements are linked with pointers (no need to be continuous)

Linked List

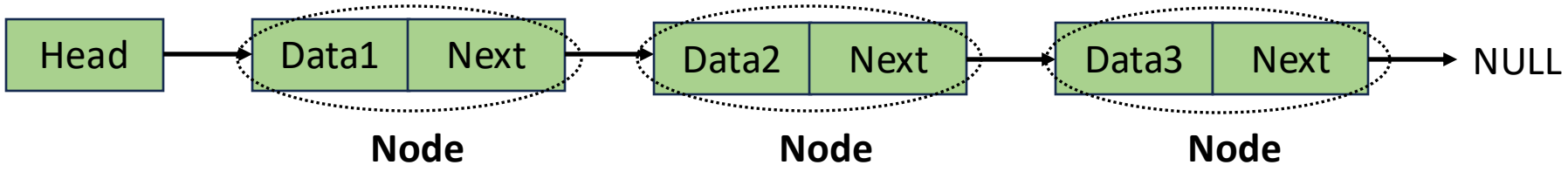
- Another linear data structure, allow dynamic space allocation
- The elements are linked with pointers (no need to be continuous)



https://v.youku.com/v_show/id_XNTEzOTM1NTExNg==.html?source=baidu&refer=sousuotoufang_market.qrwang_00002944_000000_QJFFvi_19031900

Linked List

- Implementation: creation




```
Class Node:
    next = None
    data = None

u3 = Node(next=None, data=3)
u2 = Node(next=u3, data=2)
u1 = Node(next=u2, data=1)
head = u1
```

Linked List

List operations with linked list

- Access the k-th element: O(n)
- Search for a given element
- Delete an element
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

```
Supposing the length is larger than k
p = head
for i in range(k-1):
    p = p.next
print(p.data)
```

Linked List

List operations with linked list

- Access the k-th element: O(n)
- Search for a given element: O(n)
- Delete an element
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

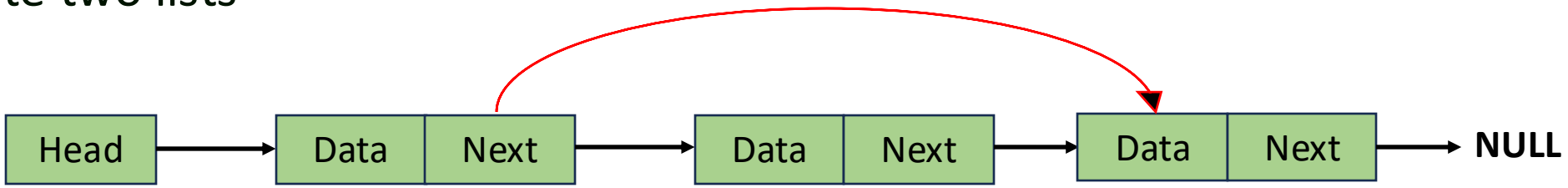
```
p = head
while p is not None:
    if p.data == target:
        break
    p = p.next
return p
```

Linked List

List operations with linked list

- Access the k-th element: O(n)
- Search for a given element: O(n)
- Delete an element: O(1)
- Insert an element at certain position
- Append a new element at the end
- Concatenate two lists

```
if p.next is not None:
    u = p.next.next
    del p.next
    p.next = u
```

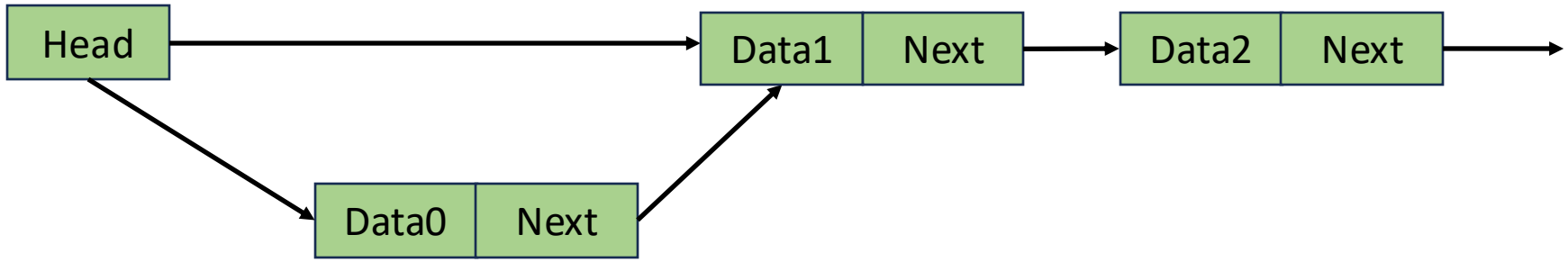


Linked List

List operations with linked list

- Access the k-th element: O(n)
- Search for a given element: O(n)
- Delete an element: O(1)
- Insert an element at certain position: O(1)
- Append a new element at the end
- Concatenate two lists

```
new_u = Node(data=0, next=p.next)
p.next = new_u
```

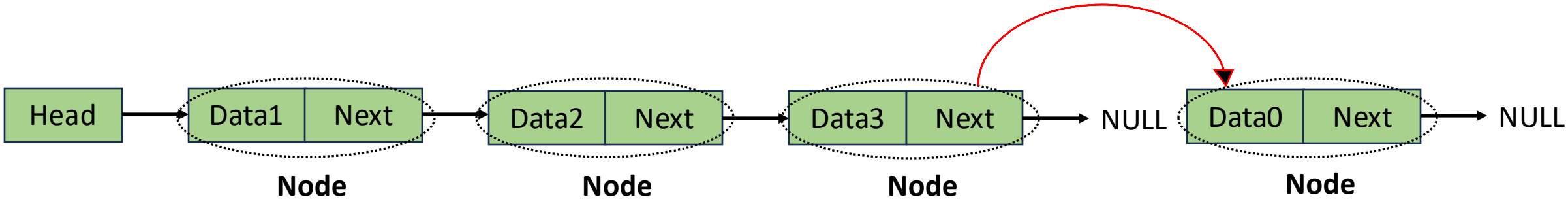


Linked List

List operations with linked list

- Access the k-th element: O(n)
- Search for a given element: O(n)
- Delete an element: O(1)
- Insert an element at certain position O(1)
- Append a new element at the end: O(1)
- Concatenate two lists

```
new_u = Node(data=0, next=None)
p.next = new_u
```

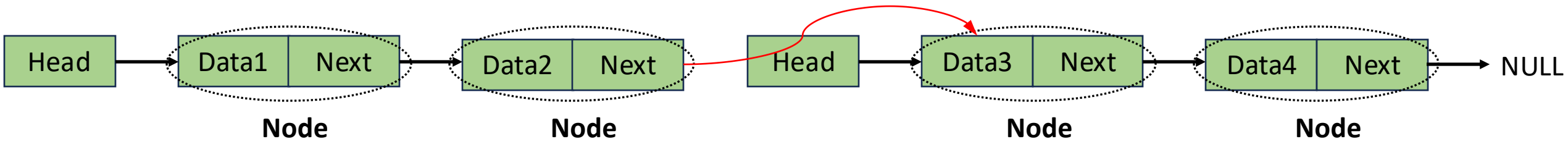


Linked List

List operations with linked list

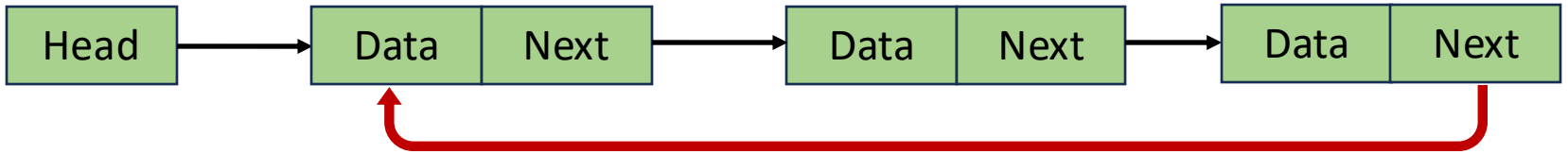
- Access the k-th element: O(n)
- Search for a given element: O(n)
- Delete an element: O(1)
- Insert an element at certain position O(1)
- Append a new element at the end: O(1)
- Concatenate two lists: O(1)

Once you know the tail of the first list
tail1.next = head2

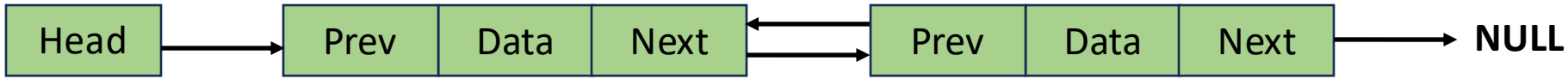


More Implementations

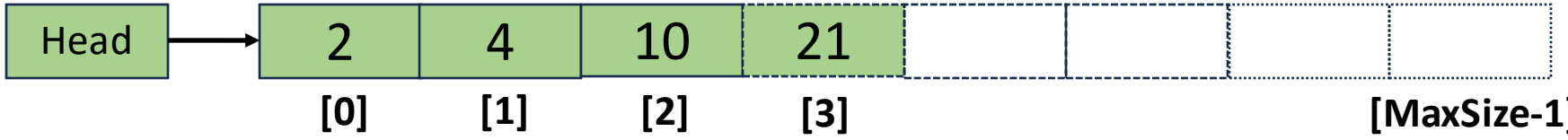
- Circular list: the next pointer of the last element points to the first element

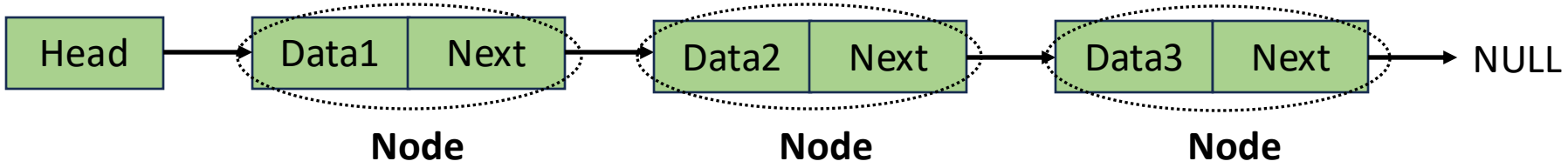


- Doubly Linked List: Each node also points to the previous node



Summary



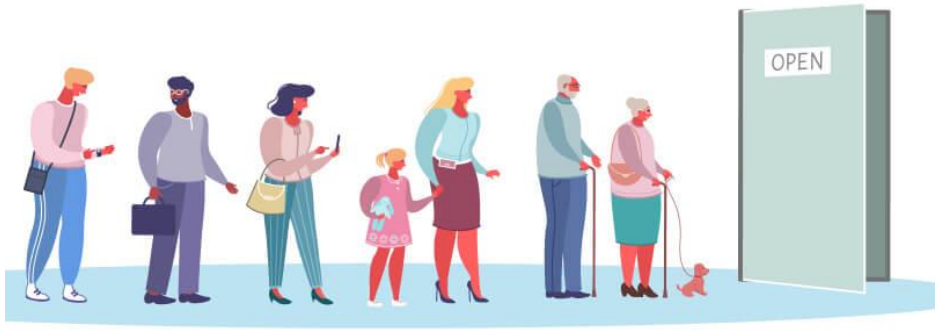


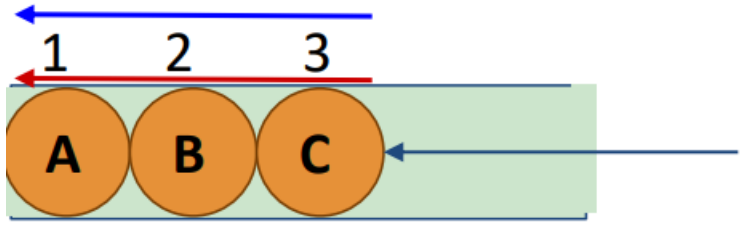
Operation	Linked List	Array
Indexing	O(n)	O(1)
Search	O(n)	O(n)
Delete	O(1)	O(n)
Insert	O(1)	O(n)
Append	O(1)	O(1)
Concatenate	O(1)	O(n)

Queues and Stacks

Queue

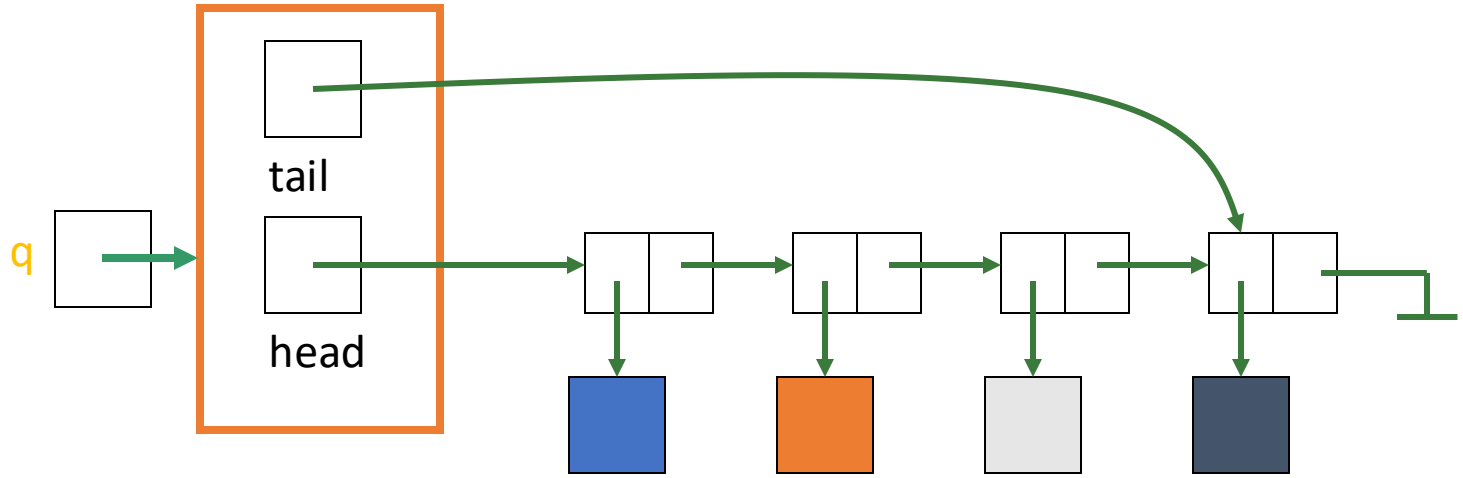
- A dynamic collection that ordered by the adding time: First In First Out (FIFO)
- Example: patients waiting in line in the Hospital ER
  - A new patient arrives - new elements pushed to the tail
  - The doctor calls for the next patient - existing elements popped from the head





Implementation (Linked List)

- Push - add a new node at the tail: O(1)
- Pop - get the value of head and remove it: O(1)
- Update head and tail pointers: O(1)

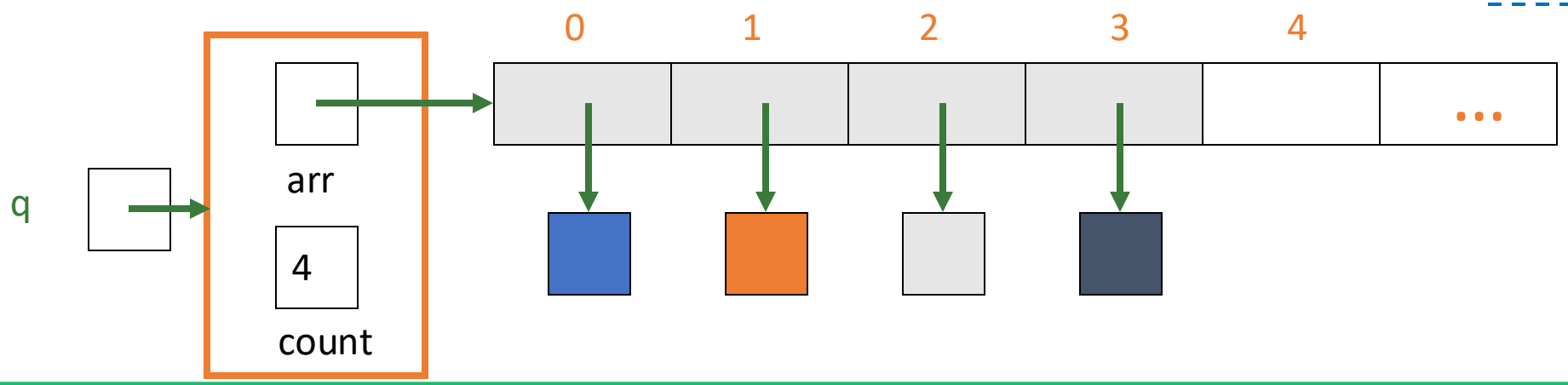



```
push(ele):
    u = Node(data=ele, next=None)
    tail.next = u
    tail = u

pop():
    ret = head
    head = head.next
    return ret.data
```

Implementation (Array)

- Simple Implementation
  - head index 0
  - Record the number of elements (i.e., tail is arr[count – 1])
- Push - add a new element and update counter: O(1)
- Pop - delete arr[0]: O(n)

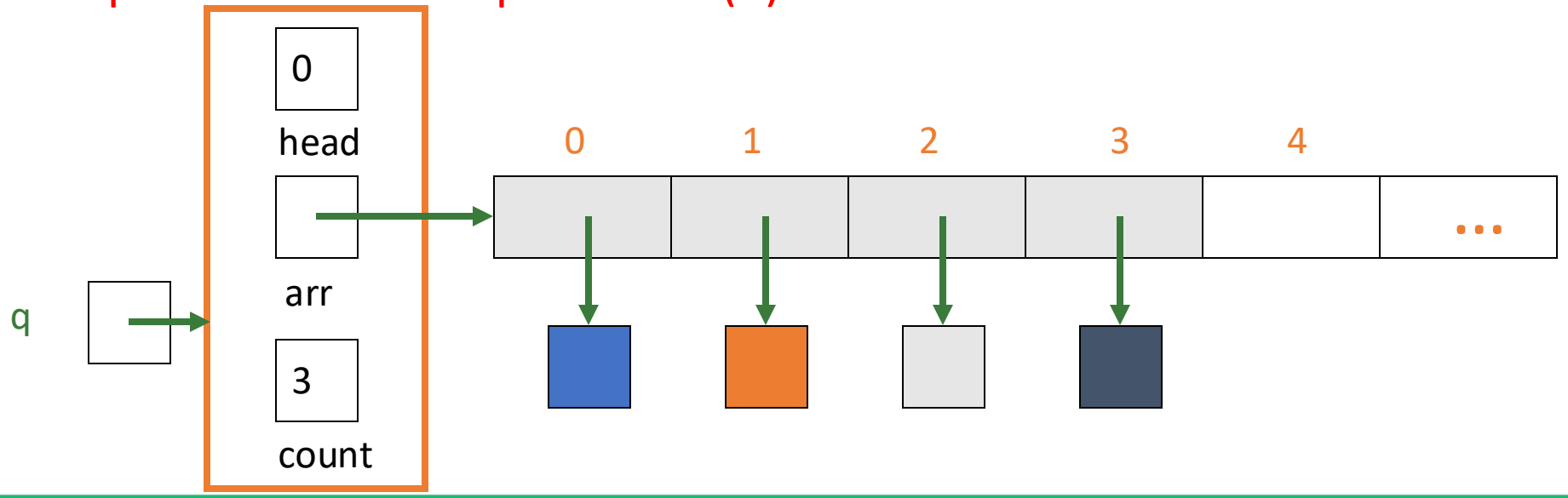
```
push(ele):
    arr[count]=ele
    count+=1

pop():
    ret = arr[0]
    for u: 1 to count-1:
        arr[u-1] = arr[u]
```



Implementation (Array)

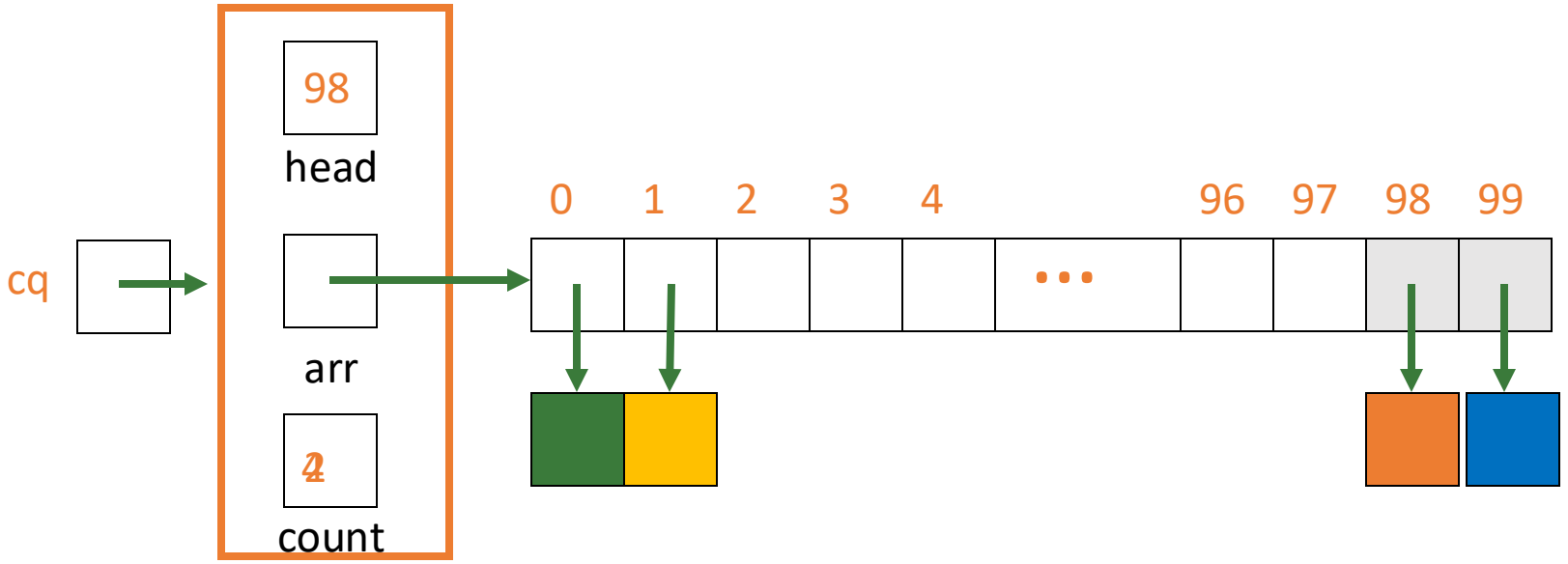
- Simple Implementation
  - Dynamic head index (initially 0)
  - Record the number of elements (i.e., tail is arr[head + count – 1])
- Push - add a new element and update counter: O(1)
- Pop - shift the head pointer: O(1)

```
push(ele):
    arr[head+count]=ele
    count+=1

pop():
    ret = arr[head]
    head += 1
    return ret
```



Implementation (Array)

- Problem: each position only used once
- Improvement: Circular Array
- head = (head+1) % arr.length
- tail: (head + count – 1) % arr.length

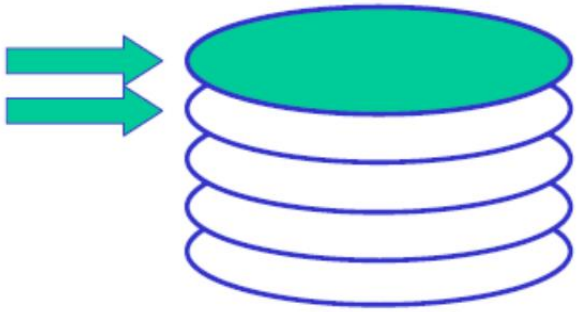


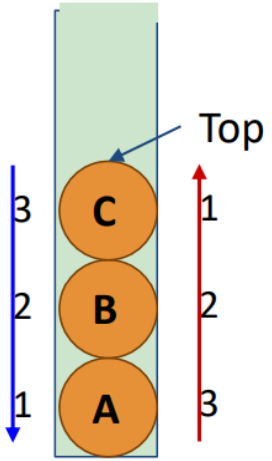
Queue: Summary

	Linked List Implementation	Array Implementation
Enqueue(push)	O(1)	O(1)
Dequeue(pop)	O(1)	O(1)
	Unlimited storage	Need resizing when full

Stack

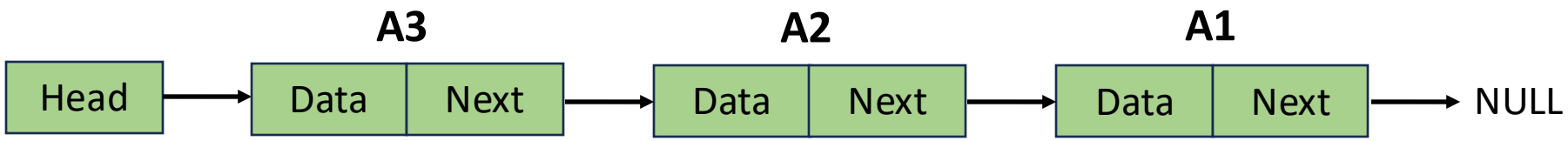
- A dynamic collection that ordered by the adding time: Last In First Out (LIFO)
- Example: a pile of plates
  - Put a new plate - **Pushing** a new element to the top
  - Use a plate - **Popping** the element from the top





Implementation (Linked List)

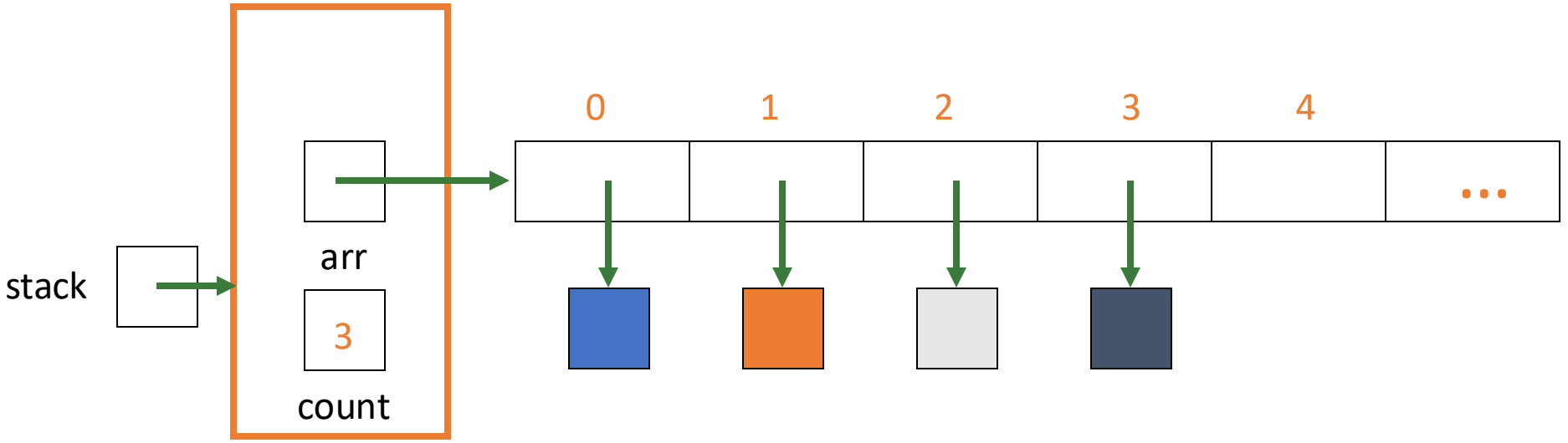
- Push and pop at the head: O(1)




```
push(ele):
    u = Node(data=ele, next=head)
    head = u

pop():
    p = head
    head = head.next
    return p.data
```

Implementation (Array)

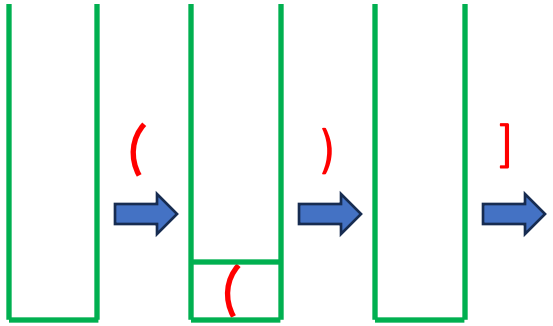
- Count the number of elements in the stack
- arr[count-1] is the top
- Add the element at arr[count]: O(1)
- Pop the element from arr[count - 1]: O(1)




```
push(ele):
    arr[count]=ele
    count+=1

pop():
    ret = arr[count-1]
    count -= 1
    return ret
```

Example: Bracket Balance Check

- Problem: Given an expression string, write a program to examine whether the pairs and the orders of “{”, “}”, “(”, “)”, “[”, “]” are correct in the given expression.
  - [()]{}{[()()]()} - correct, [(]) – not correct
- Solution: stack
  - Opening bracket: push to the stack
  - Closing bracket: (1) check if it matches the top character, (2) pop the top character
  - Unbalanced: (1) the top does not match the current closing or is empty, (2) not empty in the end



()]： Not correct

Example: Expression Parsing

- Problem: given a simple arithmetic expression containing numbers and arithmetic operators (+ - * /), output the result

Example Input

3 + 2 * 5 – 1

Example Output

12

- Key: dealing with operator precedence

Example: Expression Parsing

3 + 2 * 5 - 1

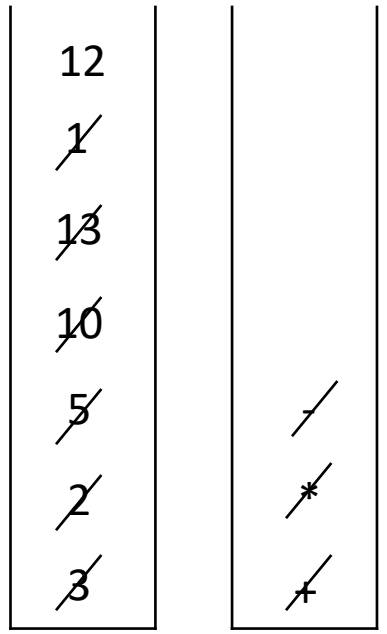
Solution with stack

1. Maintain an operator stack and a number stack
2. Traverse token from left to right

- **Number:** push to number stack
- **Operator:**
  - If top operator's precedence larger than or equal to the current one: pop and calculate
  - Push the token operator onto the operator stack

3. Finally, operators in stack are in precedence decreasing order

   Sequentially pop and calculate until the operator stack is empty

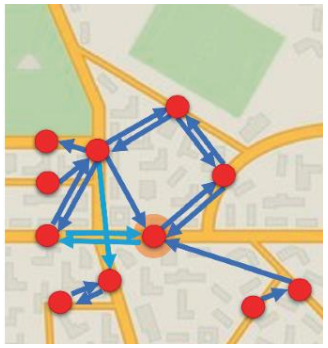


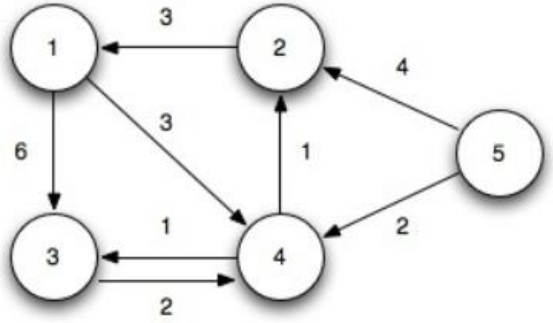
Graphs and Trees

Graph

Graph $G = (V, E)$: A non-linear data structure consisting of **vertices** and **edges**.

- **Vertices V:** fundamental units of the graph, **Edges E:** connection between two nodes

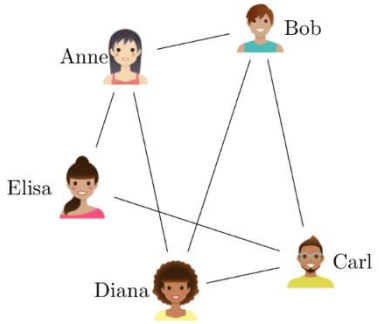




Road network (distance/cost)

Weighted: Weight associated with edges

Directed: each edge has a direction



Social network (friendships)

Unweighted: No weight associated with edges

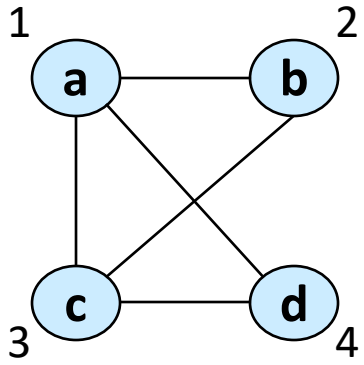
Undirected: edges have no direction

Graph Representation

- **Adjacency Matrix:** |V|*|V| matrix, entries representing the edge (weights)

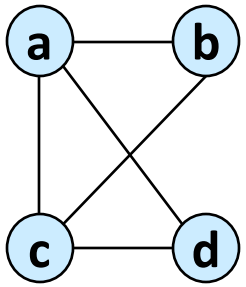
$$M[i,j] = \begin{cases} 1/weight & \text{if } (i,j) \in E \\ 0/inf & \text{otherwise} \end{cases}$$

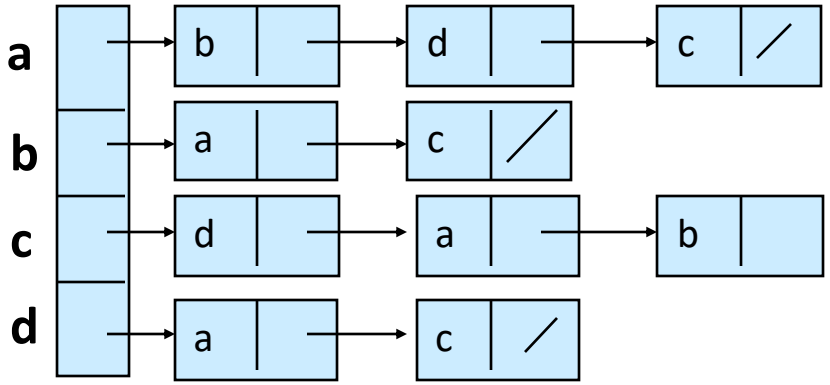
M = M^T for undirected graphs.



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

- **Adjacency List** (each vertex has a list of vertices to which it is adjacent)

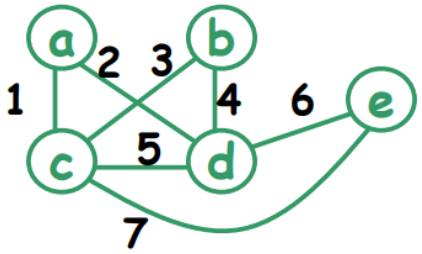




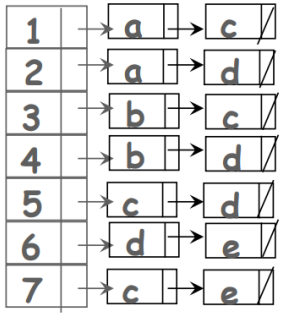
Weights can be also stored

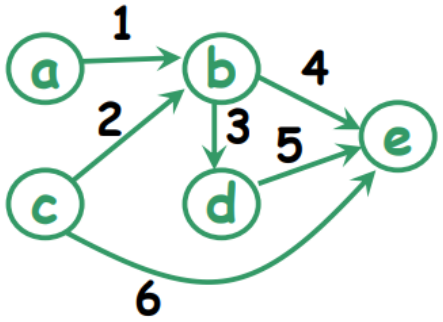
Graph Representation

- **Incidence Matrix:** |E|*|V| matrix, entries indicates ends of the edge
  - *Undirected* $M[i,j] = \begin{cases} 1 & e_i \ links \ v_j \\ 0 & \text{otherwise} \end{cases}$
  - Directed: $M[i,j] = \begin{cases} 1 & v_j \ is \ e_i's \ source \\ -1 & v_j \ is \ e_i's \ target \\ 0 & \text{otherwise} \end{cases}$
  - Weighted? depending on the application

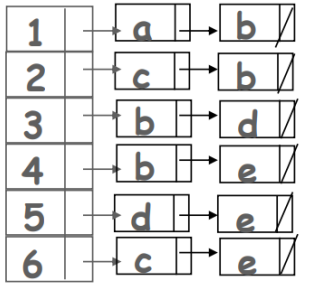


	a	b	c	d	e
1	1	0	1	0	0
2	1	0	0	1	0
3	0	1	1	0	0
4	0	1	0	1	0
5	0	0	1	1	0
6	0	0	0	1	1
7	0	0	1	0	1





	a	b	c	d	e
1	1	-1	0	0	0
2	0	-1	1	0	0
3	0	1	0	-1	0
4	0	1	0	0	-1
5	0	0	0	1	-1
6	0	0	1	0	-1



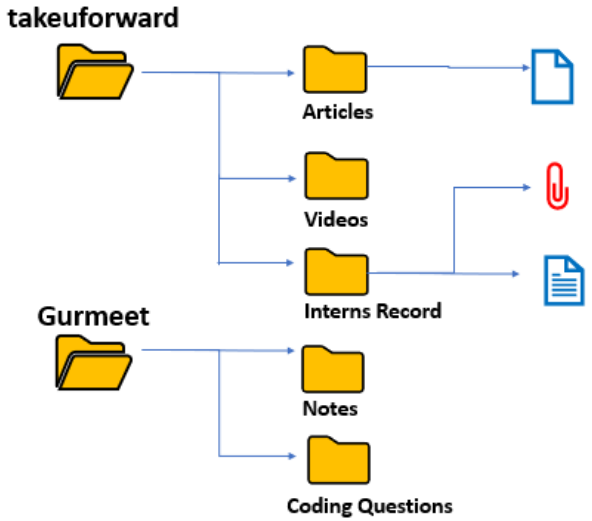
Labels: edge number

- **Incidence List:** a list of vertices for each edge

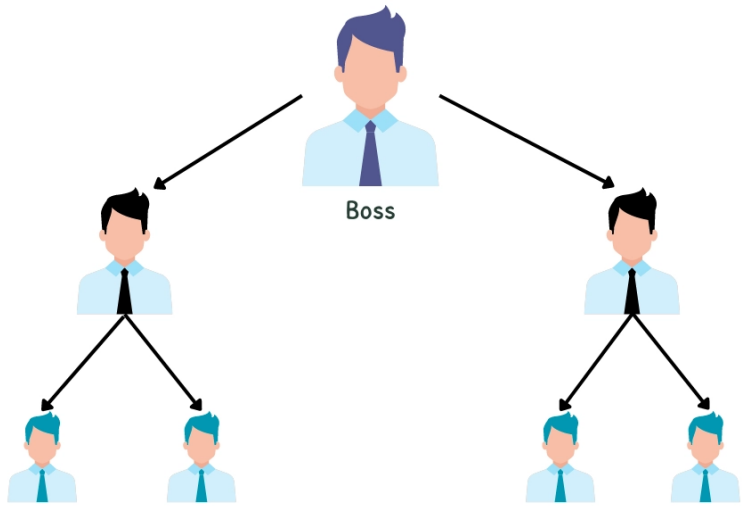
Tree

A hierarchical structure that consists of nodes connected by edges, a special directed graph

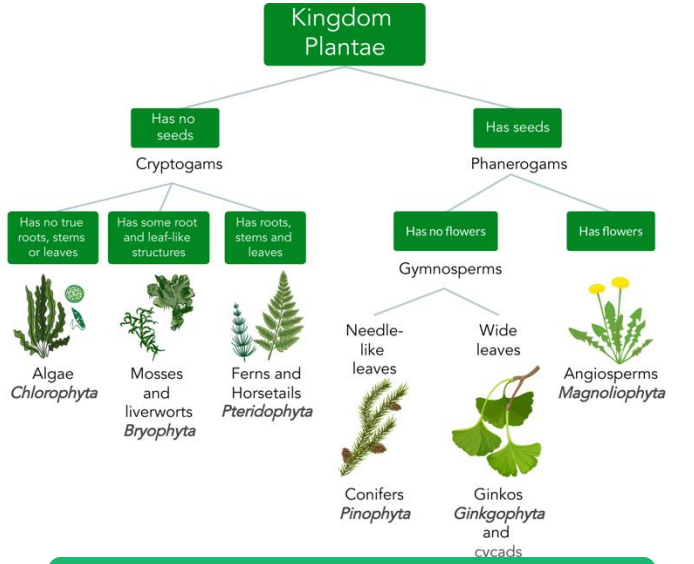
- Each node can be connected to many children, but must be connected to exactly one parent, except for the root node (no parent) – exactly n - 1 edges
- Sometimes we omit the direction



Computer file system



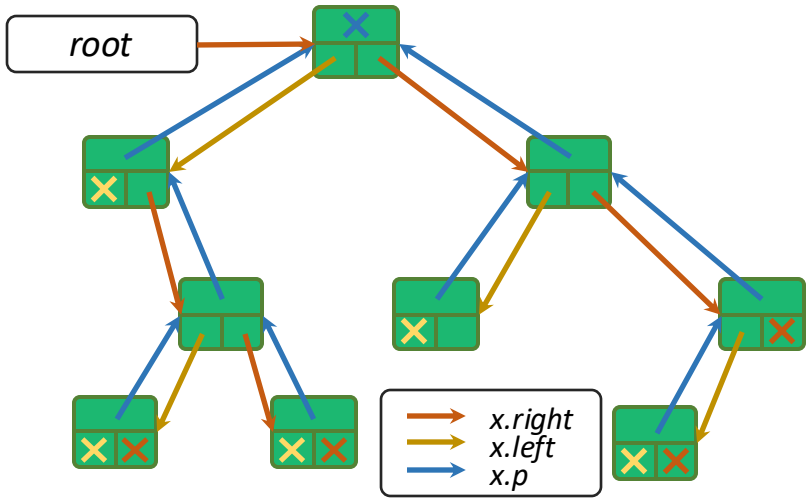
Company organizational structure



Species Taxonomy

Binary tree Representation

- $T.root$ points to the root node. $T.root = NIL$ means the tree is empty.
- Each node: attributes $p, left, right$ store pointers to the parent, left child, and right child of each node. $x.p = NIL$ when x is root. $x.left = NIL$ when x has no left child, and similarly for the right child.

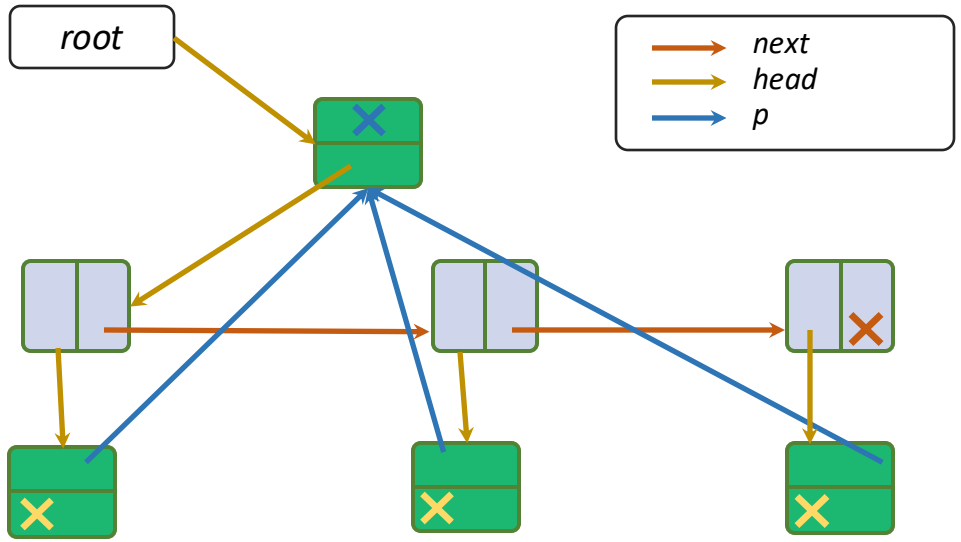



```
Class Node{
   data, p, left, right
}
u1 = Node(data='A', p=None, left=None, right=None)
u2 = Node(data='B', p=u1, left=None, right=None)
u3 = Node(data='C', p=u1, left=None, right=None)
u1.left=u2; u1.right=u3
root=u1
```

Multi-way tree Representation

Store nodes in a list (e.g., linked list)

```
Class Node{
   data, p, head
}
Class ListNode{
   node, next
}
u1 = Node(data='A', p=None, head=None)
u2 = Node(data='B', p=u1, head=None)
u3 = Node(data='C', p=u1, head=None)
l1_u3 = ListNode(u3, next=None)
l1_u2 = ListNode(u2, next=l_u3)
u1.head=l1_u2
root=u1
```

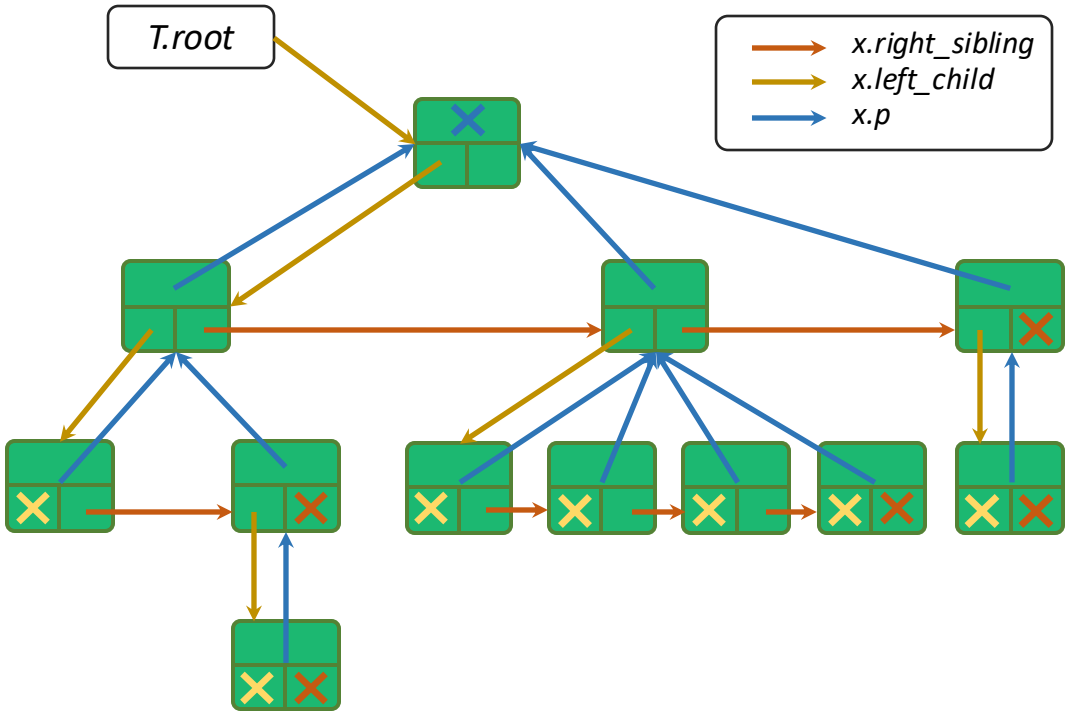


$O(n)$ space for any n-node rooted tree!

Multi-way tree Representation

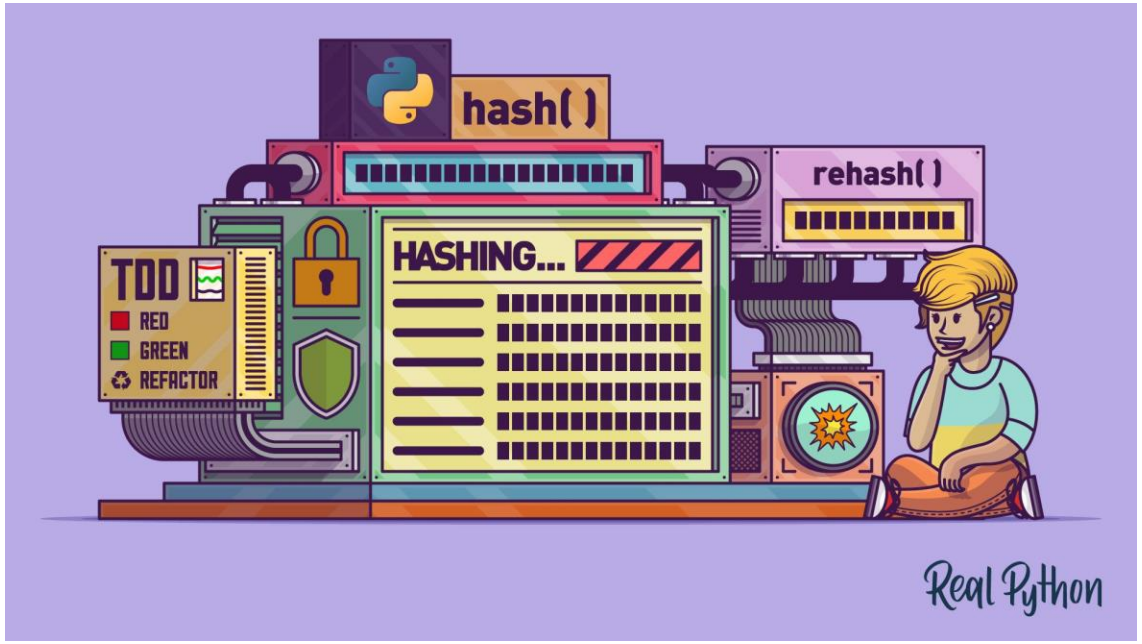
More efficient: merge linked list nodes and tree nodes

- $x.left\_child$ points to the leftmost child of node x
- $x.right\_sibling$ points to the sibling of x immediately to its right.
- If node x has no children, then $x.left\_child = NIL$, and if node x is the rightmost child of its parent, then $x.right\_sibling = NIL$.



$O(n)$ space for any n-node rooted tree!

Hash Table



Requirement for an ADT

A dynamic collection

- Store information based on *keys*
  - key: identifier of an element
  - value/satellite data: associated information
  - (WLOG, we omit value)
- Support search, insert, and delete operations
- Example: '**dict**' in Python

Simple linear list?

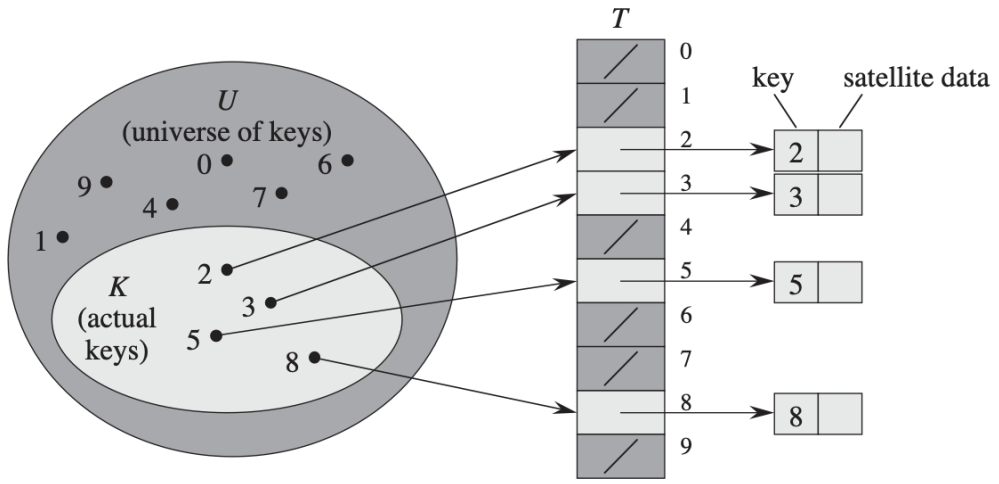
- O(1) for insert
- O(1)/O(N) for delete
- $O(N)$ for search

Direct Addressing Table

Case 1: The keys of elements are drawn from a (not large) universe $U = \{0, 1, \ldots, m-1\}$

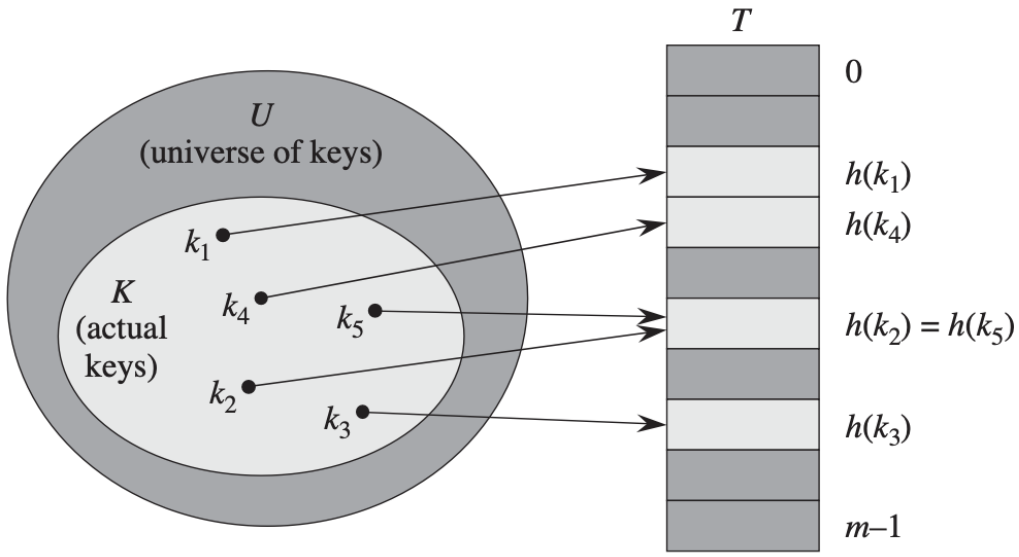
- Use an array with m slots to represent a dynamic collection of these elements
- Elements stored in slots according to their key. $T[k] = NIL$ if no element with key k
- Problem: (1) Impractical for a large universe U, (2) Waste of space when K small relative to U

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

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

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



Hash Table

- Create a table with fewer slots than |U|, and use a hash function to map key k to slot h(k)
- Advantage: reduce the storage requirement to |K| while maintaining the searching for an element still O(1) in average time.
- Notice: this bound is for the *average-case time*, whereas for direct addressing it holds for the *worst-case time*.

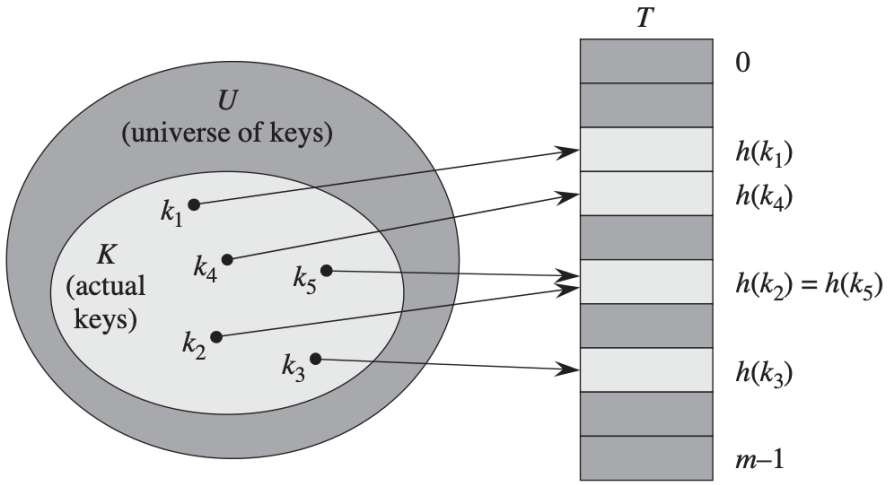


Hash Function

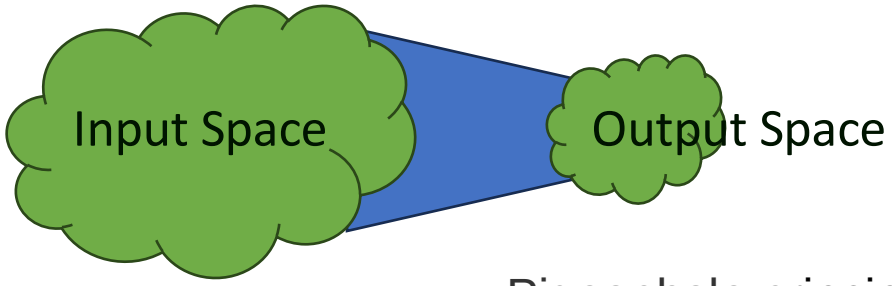
- Map data of arbitrary size to data of a fixed size
  - e.g., SHA-256: from string to 256-bit numbers
- Advantage:
  - Reduce large key space to a finite natural integer space
  - Matching without knowing the original content (e.g., password stored as hash code in the database)

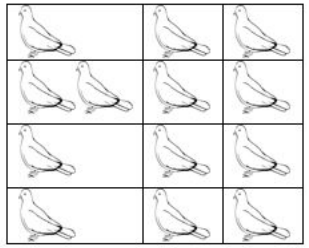
Collision

- Two keys may be projected to the same slot, which is referred to as collision



- With |U| > m, collision can be reduced but not avoidable





Pigeonhole principle: must be two keys with the same hash value

Handling Collisions: Chaining

- Organize the elements hashed to the same slot with a linked list.

CHAINED-HASH-INSERT(T, x)

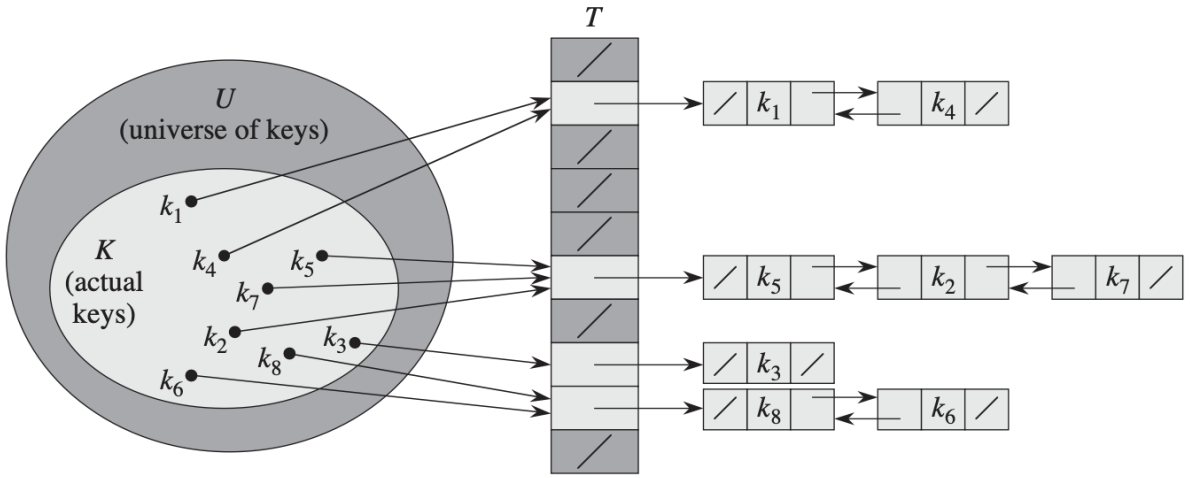
1 insert x at the head of list $T[h(x.key)]$ O(1)

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$?

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$ O(1)



Chaining: Searching Time Complexity

Number of keys: n, number of slots: m

Worst-case $\Theta(n)$: all n keys hash to the same slot, degenerate to linked list-based dynamic collection

Average-case:

- Assumption (***simple uniform hashing***): any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to

⇒All the chain has an average cost of $\boldsymbol{\alpha} = n/m$ (***load factor***) for traversing, with constant time for calculating the hash value, time complexity is $\Theta(\alpha + 1)$

⇒By setting $m = g(n)$ s.t. $n = O(g(n))$, complexity is $O(1)$

- With $f(n) = \Theta(\alpha + 1)$, there exists positive c_1, c_2, n_0, so that when $n \geq n_0$, $0 \leq c_1(\alpha + 1) \leq f(n) \leq c_2(\alpha + 1)$
- With $n = O(g(n))$, there exists positive c_3 and n_0', so that when $n \geq \mathrm{n}_0'$, $n \leq c_3 g(n)$, i.e., $\frac{n}{g(n)} = \alpha \leq c_3$
- $f(n) \leq c_2(\alpha + 1) \leq c_2(c_3 + 1)$, meaning $f(n) = O(1)$

Design a Good Hash Function

- Challenges to realize simple uniform hashing:
  - Non-independent occurrence of keys, unknown key distribution
  - Intuition: in a system, keys with similar patterns are likely to appear together
  - e.g., a compiler's symbol table stores identifiers in a program. (lala, lalala, bianliang1, bianliang2, …)

- Better hash function to make hash values independent of patterns of keys

Common Hash Functions

Interpreting keys as natural numbers, then

- **Division Method**: take the remainder of k divided by m as hash value, i.e., h(k)= k mod m
  - m should not be the power of 2, otherwise hash is to take the last several bits
  - Usually choose a prime not too close to an exact power of 2
- **Multiplication method**: Multiply k by a constant $0 < A < 1$, multiply the fractional part of kA with m and take the floor, i.e., $H(k) = \lfloor m\ (kA\ mod\ 1) \rfloor$, the value of m is not critical
- **Universal hashing**: refer to the book

h(Key)	Key
0	10000000
1	991312391
2	
3	
4	31284838274
5	2483819192715
6	747583409236
7	
8	
9	4328289129

Example: h(key) = key%10

Handling Collisions: Open Addressing

Another solution for handling collisions without using additional lists

- Key idea: successively probe the hash table until finding an empty slot to put the key
- Generic formula of hash function
  - Initial h(k): project the key k to a hash value
  - Probing h(k, i): project the key k to a hash value at the i-th probe (i.e., the slot to be explored in the i-th try)
  - Probe sequence: <h(k, 0), h(k, 1), …, h(k, m-1) >

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Handling Collisions: Open Addressing

Linear probing: probes with a constant step

- h(key, i) = h(key) + i * step % m
- Prob sequence: h(key) , (h(key) + step) % m, (h(key) + 2*step) % m, …

- Example: h(str) = str[0], set step = 1
  - add ‘age’

Slot	Value
0	ale
1	bay
2	age
3	
4	egg
5	
6	
7	home
8	
9	

loc: h(age) = 0

(loc + step) % 10 = 1

(loc + step*2) % 10 = 2

Handling Collisions: Open Addressing

Linear probing: probes with a constant step

- h(key, i) = h(key) + i * step % m
- Prob sequence: h(key) , (h(key) + step) % m, (h(key) + 2*step) % m, …

- Example: h(str) = str[0], set step = 1
  - Search ‘age’
  - Search ’apple’: stop at empty

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Slot	key
0	ale
1	bay
2	age
3	
4	egg
5	
6	
7	home
8	
9	

loc: h(apple) = 0

(loc + step) % 10 = 1

(loc + step*2) % 10 = 2

(loc + step*3) % 10 = 3

Handling Collisions: Open Addressing

Linear probing: probes with a constant step

- h(key, i) = h(key) + i * step % m
- Prob sequence: h(key) , (h(key) + step) % m, (h(key) + 2*step) % m, …

- Example: h(str) = str[0], set step = 1
  - Delete ‘ale’ and search ‘age’

Slot	key
0	
1	bay
2	age
3	
4	egg
5	
6	
7	home
8	
9	

Cannot find age!

loc: h(age) = 0

Handling Collisions: Open Addressing

- Improve: record the status as occupied, empty, or deleted
- Stop when encountering empty
- Problem: Not efficient when deletion frequently occur
  - Probed slots become long as the system continuously running
  - All the slots can gradually become ‘deleted’
- Usually use chaining in this case
- Notice: we do not consider deleting for ease of presentation

Slot	Key	Status
1	ale	occupied
2	bay	occupied
3	age	occupied
4		
5	egg	occupied
6		
7	gift	occupied
8	home	occupied
9		
10		
11		
12		

Handling Collisions: Open Addressing

Primary clustering problem for linear probing

- The average search time increases with continuously occupied slots increase
- e.g., when step=1, for a slot with i previous slots occupied, the probability of it get required by a random key become $\frac{i+1}{m}$ (with *simple uniform hashing assumption*)

Slot	Key
0	ale
1	bay
2	age
3	acre
4	egg
5	
6	gift
7	home
8	
9	
10	
11	

keys starting with ‘a’ to ‘f’ all prob this slot

Handling Collisions: Open Addressing

Eliminate the problem: more complex probing

- quadratic probing: $h(k,i) = (h(k) + c_1 i + c_2 i^2) \, mod \, m$
- Problem (secondary clustering): If H(x)=H(y), then still the same probe sequence

Seems natural? But can be avoided.

- Notice: only m possible probe sequences are used for the entire U
- Ideally, the keys in U uniformly distribute in the $m!$ probe sequences (**uniform hashing assumption**)

Analysis of open-address hashing

- In open addressing, each slot has at most one element, thus $\alpha = \frac{n}{m} < 1$
- Uniform hashing assumption: the probe sequence is equally likely to be any permutation of <1, …, m>.
- ***Theorem:*** Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in an unsuccessful search (X) is at most $1/(1-\alpha)$, assuming uniform hashing.
  - $E[X] = \sum_{i=1}^{\infty} i\Pr\{X = i\} = \sum_{i=1}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$
  - $\Pr\{X \geq i\}$: The first i - 1 picked slots must be occupied
    - Uniform hashing: each probe step can be regarded as randomly picking an unused slot
    - In the j-th probe step, the probability of picking an occupied slot is $(n - (j-1))/(m-(j-1))$
    - $\Pr\{X \geq i\} = \prod_{j=1}^{i-1} \frac{n-(j-1)}{m-(j-1)} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$
  - $E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$ (set proper m to make it small)
- ***Corollary:*** Inserting element needs to go through an unsuccessful search, so also $\frac{1}{1-\alpha}$

Analysis of open-address hashing

- ***Theorem:*** Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a *successful search* is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$, assuming (1) uniform hashing and that (2) each key in the table is equally likely to be searched for.

  - During searching for a key, the probe sequence is the same as it when `inserting it.
  - The slots probed before finding the key are the ones occupied when the key was inserted
  - For the $(i+1)$st key inserted in the table, the expected search probes is at most $\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$ (according to the corollary)
  - The expected number of probes for all keys: $\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}$
  - Since $\frac{1}{k}$ monotonically decreasing, we have $\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \le \frac{1}{\alpha}\int_{m-n}^{m}\left(\frac{1}{x}\right)dx = \frac{1}{\alpha}\ln\frac{\mathrm{m}}{\mathrm{m-n}} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

Handling Collisions: Open Addressing

- Double hashing: use more probe sequences
- $h(k,i) = (h_1(k) + ih_2(k))\ mod\ m$
  - $h_1(k)$ - sets position, $h_2(k)$ - defines probe step size
- To probe the entire table: $h_2(k)$ should be relatively prime to m, e.g.,
  1. $m = 2^p$ and h_2 outputs odd numbers
  2. m prime and $h_2(k) < m$ (e.g., mod a smaller value)
- m^2 possible probe sequences in total (h_1 and h_2 both have m possible outputs)
- Reduce collision risk of probe sequence

Summary

- Data Structure
- Array and linked lists
- Queues and Stacks
- Graphs and Trees
- Hash Tables

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures