Recap of ADT and Stacks

- Abstract Data Type
  - Stack: two basic operations: push and pop
  - Stack with an unlimited amount of entries, of arbitrary types
  - What would it be like if we implemented a stack using Python's built in "list"?

Recap of Stack

- Stack: two basic operations: push and pop
- Primitive Data Structure implementation using the built in "list" datatype of Python
  - Good news: Able to store an unlimited amount of items: The container kept growing
    - Notion of memory manager (Capacity and actual number of elements)
  - Bad news: We had to worry about the extra time involved in push. It was clear that this was not an O(1) type operation

Recap of Stack

- Abstract Data Type: Stack
- Using the built in "list" datatype of Python
- Arrays, std::vector and Python list
  - Wanted to avoid hardcoded numbers like int a[100000]
- vector abstract data type:
  - Access is O(1): A[i]
  - A.push_back(): Insert at end would require us to spend more than O(1) time (but only sometimes)
  - Random insert is at least O(n) (may involve asking for memory also)

Recap of Stack

- A.push_back(): Insert at end would require us to spend more than O(1) time (but only sometimes)
- Amortized complexity: Can we mathematically quantify this?
  - We showed that inserts at end (or stack push equivalently) took O(1) time
- Stack Implementation
- Amortized complexity revisited

Recap of Stack

- Amortized complexity: Can we mathematically quantify this?
  - We should that inserts at end (or stack.push() equivalently took O(1) time
- Stack Implementation
- Amortized complexity revisited

Recap of Stack

- Amortized complexity: Can we mathematically quantify this?
  - We should that inserts at end (or stack.push() equivalently took O(1) time
- Stack Implementation
- Amortized complexity revisited
  - Aggregate method
  - Potential method
  - Accounting method

Recap of Stack

- Amortized complexity: Can we mathematically quantify this?
- Stack Implementation
- Amortized complexity revisited
  - Aggregate method
  - Potential method
  - Accounting method
- Use of the stack data structure in “real” problems
  - Arithmetic expression

Background and Motivation

- Notion of ADT
- Stack ADT, std::vector
- Although there is std::stack, there is no stack object per-se built into python out of the box
- Three problem areas
  - The length (capacity) of array or stack is too large
  - Amortized bounds are inappropriate in real-time situations
  - (vector) Insertions and deletions in an interior position are expensive

Agenda

- Linked allocation

Other Basic ADT

- Stack
- Queue
  - Used all the time in operating systems (e.g. scheduling of processes)
- Deque
  - Generalization of the stack and the queue (and sometimes useful just by itself)

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Example

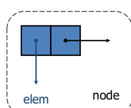
Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.is_empty()	False	[3]
Q.dequeue()	3	[ ]
Q.is_empty()	True	[ ]
Q.dequeue()	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first()	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue()	7	[9, 4]

The Deque ADT

- The Deque ADT stores arbitrary objects and generalizes stack/queue
- Insertions and deletions can either be at the front or at the rear
- Main operations:
  - add_first(object): inserts an element at the front
  - object delete_first(): removes and returns the element at the front
  - add_last(object): inserts an element at the end
  - object delete_last(): removes and returns the element at the end
- Auxiliary operations:
  - object first(): returns the element at the front without removing it
  - object last(): returns the element at the emd without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of delete_last or delete_first EmptyDequeException

The Node Class for List Nodes

```
public class Node {
  // Instance variables:
  private Object element;
  private Node next;
  /** Creates a node with null references to its element and
    next node. */
  public Node() {
    this(null, null);
  }
  /** Creates a node with the given element and next node. */
  public Node(Object e, Node n) {
    element = e;
    next = n;
  }
```

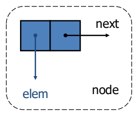


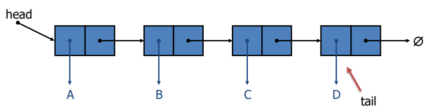
The Node Class for List Nodes

```
public class Node {
  // Instance variables:
  private Object element;
  private Node next;
  /** Creates a node with null references to its element and next node. */
  public Node() {
    this(null, null);
  }
  /** Creates a node with the given element and next node. */
  public Node(Object e, Node n) {
    element = e;
    next = n;
  }
  // Accessor methods:
  public Object getElement() {
    return element;
  }
  public Node getNext() {
    return next;
  }
  // Modifier methods:
  public void setElement(Object newElem) {
    element = newElem;
  }
  public void setNext(Node newNext) {
    next = newNext;
  }
}
```

Singly Linked List

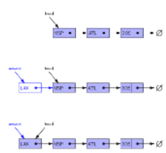
- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





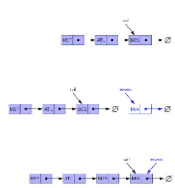
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



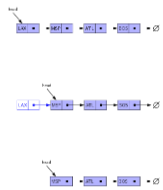
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null (or None)
4. Have old last node point to new node
5. Update tail to point to new node



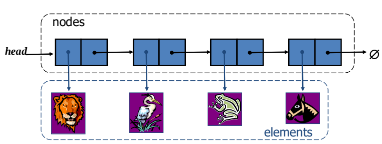
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



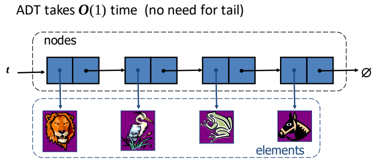
Stack as a Linked List

- We can implement a stack with a singly linked list



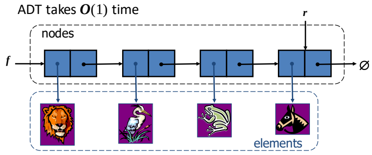
Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time (no need for tail)



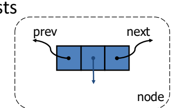
Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Recap Motivation

- Three problem areas
  - The length (capacity) of array or stack is too large
  - Amortized bounds are inappropriate in real-time situations
  - (vector) Insertions and deletions in an interior position are expensive!
- Solution: implement doubly-linked lists

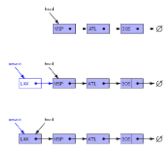


Agenda

- Linked allocation vs Array Allocation
- Implementation aspects

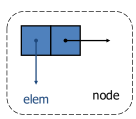
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



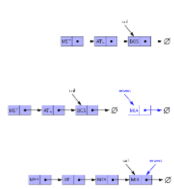
The Node Class for List Nodes

```
public class Node {
  // Instance variables:
  private Object element;
  private Node next;
  /** Creates a node with null references to its element and
    next node. */
  public Node() {
    this(null, null);
  }
  /** Creates a node with the given element and next node. */
  public Node(Object e, Node n) {
     element = e;
     next = n;
  }
```



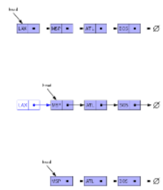
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null (or None)
4. Have old last node point to new node
5. Update tail to point to new node



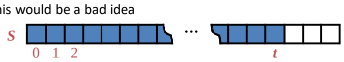
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



Array-based Queue

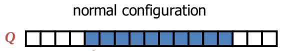
- A simple way of implementing the Stack ADT uses an array
  – We add elements from left to right
  – A variable keeps track of the index of the top element
  – We use the append() method of list to push
  – We use the pop() operator of list
- For a Queue we could use the pop(0) to remove from the front
  – This would be a bad idea



Array-based Queue

- Use an array of size N
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty
- This is problematic since with time the space consumed will be proportional to the number of queries, not the number of items

normal configuration



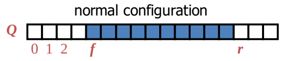
Queue in Python

- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - _front: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).

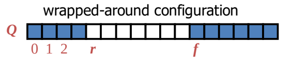
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

normal configuration



wrapped-around configuration

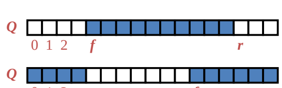


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

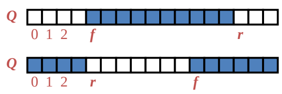
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

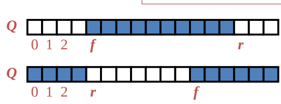
```
Algorithm enqueue(o)
  if size() = N − 1 then
    throw FullQueueException
  else
    Q[r] ← o
    r ← (r + 1) mod N
```



Queue Operations (cont.)

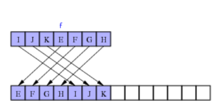
- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```



Queue Resize

- Resize is more problematic compared to a stack
  - The modular arithmetic is dependent on the capacity of the queue
  - Set f to be 0 (which is what it was when we started the queue operation)



Array-Based sequences

- Arrays provide O(1)-time access to an element based on an integer index
  - Need to traverse a linked-list from the head
- Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array
  - For example, adding an element requires us to create a node
- Array-based representations typically use proportionally less memory than linked structures

Link-based Sequences

- Link-based structures provide worst-case time bounds
  - As opposed to an amortized bound
- Link-based structures support O(1)-time insertion and deletions at arbitrary position
  - A loop is required to shift items when we do an insert in an array

Recap

- Remember, no one single method is best
  - Depends on what operations we want to support and in what context
- For example, the python hybrid implementation of deque uses both arrays and doubly linked lists
  - Supports worst case O(1) inserts and deletes at the end points but