stacks and queues

Tuesday, October 4, 2022

TODAY: higher-level abstractions

stacks, queues } details & implementation

searching as an application of stacks and queues
- depth-first search } stacks
- breadth-first search } queues

insert, remove } OrderedCollection — higher-level abstract data type

push, pop } Stack — last in, first out — LIFO

enqueue, dequeue } Queue — first in, first out — FIFO

} both ADTs

Why bother with these very constricted ways of handling elements?

Prevents user from doing things improperly in the context of the application.

OrderedCollection ADT

```
template <typename T>
class OrderedCollection {
  public:
    virtual ~OrderedCollection() {}; // destructor
    virtual void insert(T element) = 0;
    virtual T remove() = 0;
    virtual int getSize() = 0;
    virtual bool isEmpty() = 0;
    virtual T peek() = 0; // returns next element
                          // without removing it
};
```

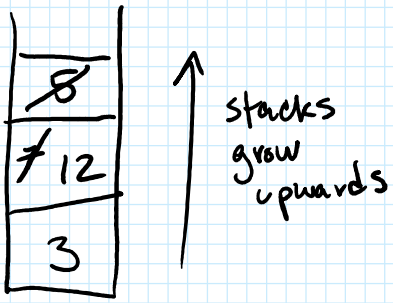
```
};
```

Stack ADT

```
template <typename T>
class Stack : public OrderedCollection<T> {
  public:
    virtual ~OrderedCollection() {};  // destructor
    virtual void push(T element) = 0;
    virtual T pop() = 0;
    virtual int getSize() = 0;
    virtual bool isEmpty() = 0;
    virtual T peek() = 0;  // returns next element
                           // without removing it
};
```

example behavior:

```
Stack.push(3)
Stack.push(7)
Stack.push(5)
Stack.peek()   => 5
Stack.pop()    => 5
Stack.pop()    => 7
Stack.push(12)
Stack.getSize() => 2
```



Queue ADT

```
template <typename T>
class Queue : public OrderedCollection<T> {
  public:
    virtual ~OrderedCollection() {};  // destructor
    virtual void enqueue(T element) = 0;
    virtual T dequeue() = 0;
    virtual int getSize() = 0;
```

```
virtual bool isEmpty() = 0;
virtual T peek() = 0; // returns next element
                      // without removing it
};
```

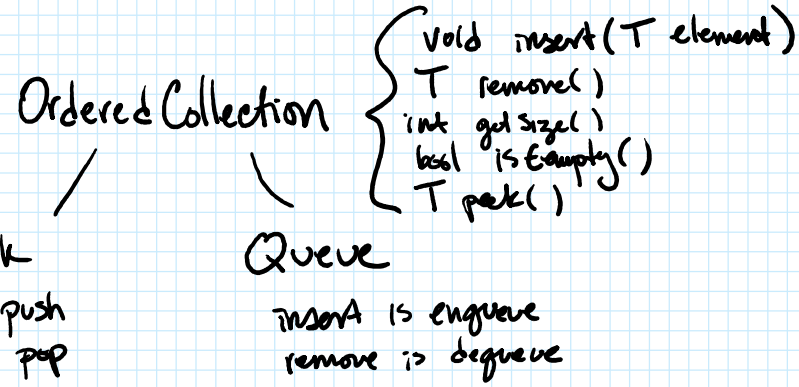
example behavior:

```
queue.enqueue(3)
queue.enqueue(7)
queue.enqueue(5)
queue.peek()        => 3
queue.dequeue()     => 3
queue.dequeue()     => 7
queue.enqueue(12)
queue.getSize()     => 2
```

front ... back

| ~~3~~ | ~~7~~ | 5 | 12 |

queues grow sideways →

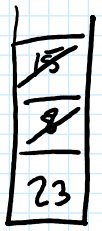


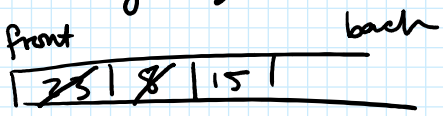
Comparing behavior of stacks & queues

Suppose we have stack* S.	Suppose we have queue* Q.
S->push(23)	Q->enqueue(23)
S->push(8)	Q->enqueue(8)
cout << S->peek()	cout << Q->peek()
S->push(15)	Q->enqueue(15)

```
cout << S->pop()
cout << S->pop()
cout << S->getSize()
```

```
cout << Q->dequeue()
cout << Q->dequeue()
cout << Q->getSize()
```



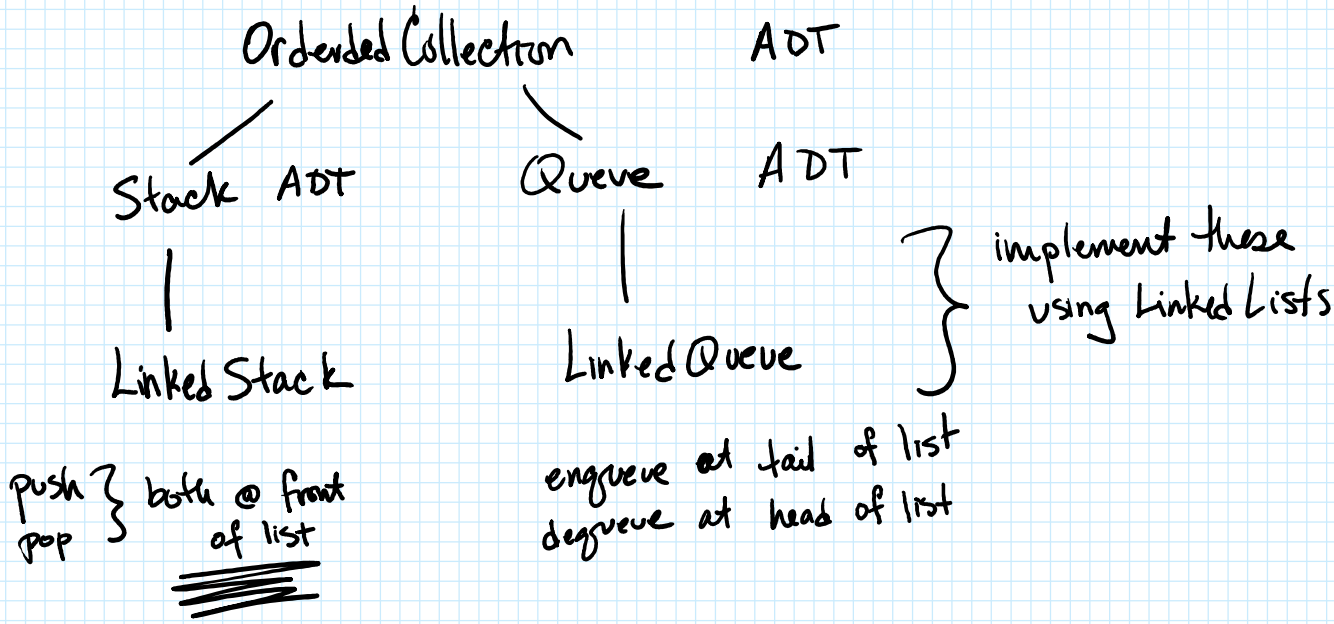


later, think about this:

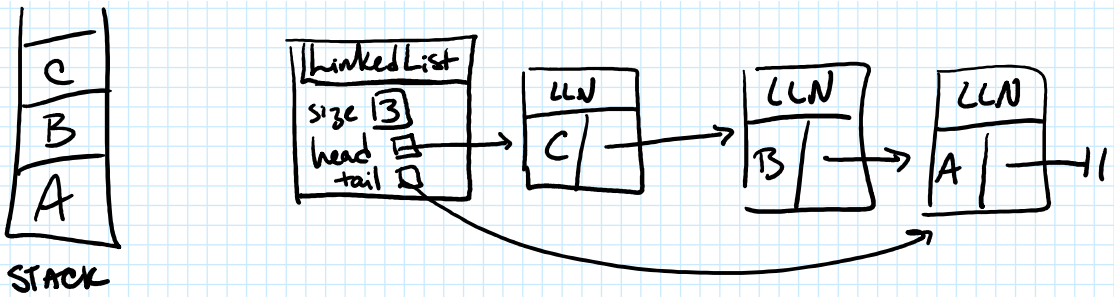
output: 8 15 8 1

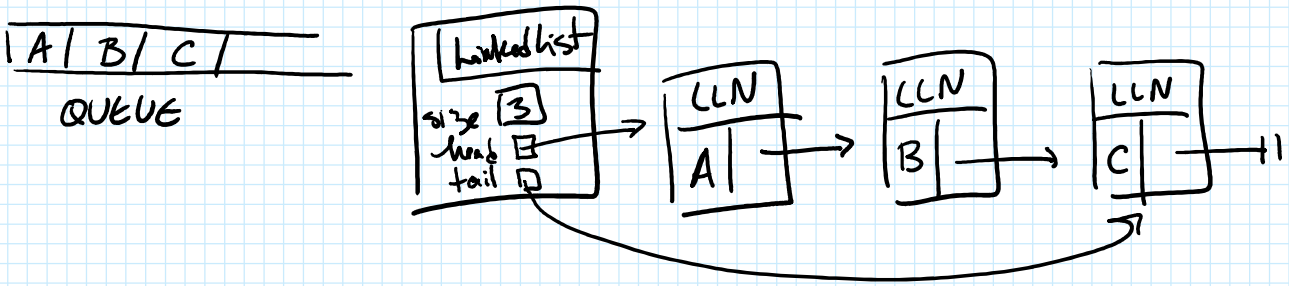
output: 23 23 8 1

How can we implement stacks and queues to make the operations run quickly?



abstract view: implementation:



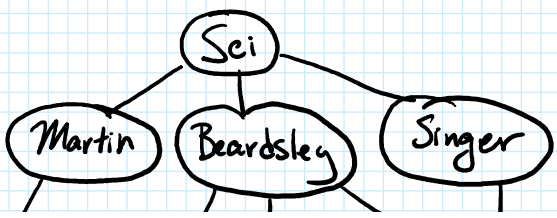


Running times

operation	LinkedStack	LinkedQueue
int getSize( )	O(1)	O(1)
bool isEmpty( )	O(1)	O(1)
T peek( )	O(1) use getFirst( )	O(1) use getFirst( )
void insert(T)	O(1) use insertAtFirst(…)	O(1) use insertLast(..)
T remove( )	O(1) use removeFirst( )	O(1) use removeFirst( )

APPLICATION of STACKS & QUEUES

How to find a path from Sci to Shaq6s?



We want to search in a systematic way that makes sense (and would work on other examples).

OUTPUT should be a path from start to goal. ← keep track of how we got there using a "previous" pointer

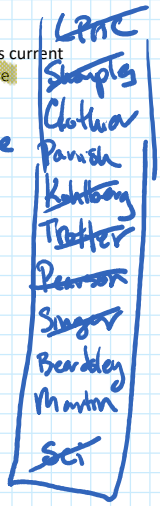
We want to make sure we only visit each location at most one time. ← for each location, keep track of whether it's been visited before

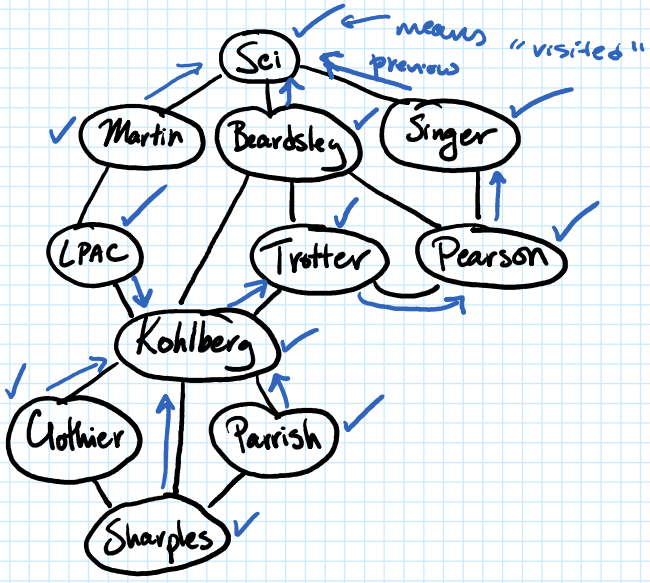
SEARCH ALGORITHM

- add start position to the data structure
- mark start as visited
- while data structure is not empty:
  - remove a position from data structure, call it current
  - if current is the goal:
    - search is complete! break
  - otherwise, for each neighbor of current:
    - if neighbor isn't visited
      - mark neighbor as visited
      - neighbor's "previous" recorded as current
      - insert neighbor into data structure

Walk-through: data structure is stack

current: ~~Sci~~ ~~Singer~~ ~~Pearson~~ ~~Trotter~~ ~~Kohlberg~~ ~~LPAC~~ Sharples





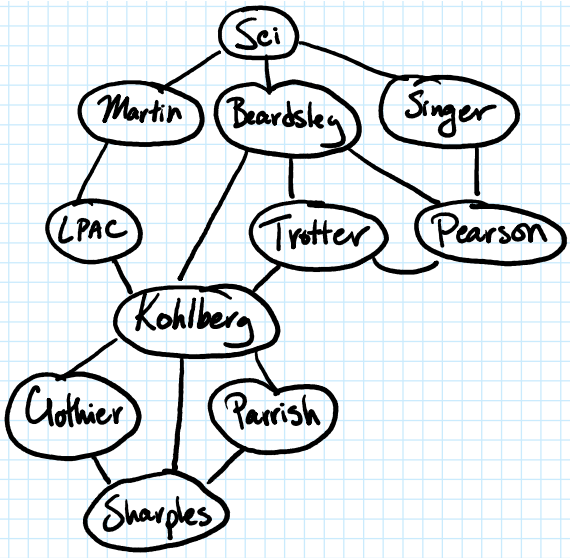
final path:

Sci, Singer, Pearson, Trotter, Kohlberg, Sharples.

SEARCH ALGORITHM

- add start position to the data structure
- mark start as visited
- while data structure is not empty:
    - remove a position from data structure, call it current
    - if current is the goal:
        - search is complete! break
    - otherwise, for each neighbor of current:
        - if neighbor isn't visited
            - mark neighbor as visited
            - neighbor's "previous" recorded as current
            - insert neighbor into data structure

walkthrough: using queue



Differences in how the search works

- when using a queue
- when using a stack