CS61B Lecture #18

:

Contest Saturday, October 14. See
/inst.eecs.berkeley.edu/~ctest/contest/
nd registration.

t week, some of the TAs will be holding 20 minute one-
ing sessions. If you would like to talk about how you're
course, get some advice about study strategies, or just
omeone to talk to, feel free to sign up! *Please do not*
as private office hours or ask the TA to debug your

cations will be available at
http://tinyurl.com/cs61b-advising.

gn up for an advising session, make sure to include some
like to talk about during the session in the description
location is not listed, it will be emailed to you before

Topics

lementation

ked: tradeoffs

sequences: stacks, queues, deques

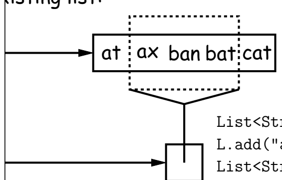
fering

d stacks

Views

A *view* is an alternative presentation of (interface to)
ct.

, the `sublist` method is supposed to yield a "view of"
isting list:




```
List<String> L = new ArrayList<String>();
L.add("at"); L.add("ax"); ...
List<String> SL = L.sublist(1,4);
```

er `L.set(2, "bag")`, value of `SL.get(1)` is `"bag"`, and
`(1,"bad")`, value of `L.get(2)` is `"bad"`.

er `SL.clear()`, `L` will contain only `"at"` and `"cat"`.

ge: "How do they *do* that?!"

Maps

d of "modifiable function:"

```
util;
ace Map<Key,Value> {
bject key);            // Value at KEY.
Key key, Value value);  // Set get(KEY) -> VALUE

-----------------------------------------
ring> f = new TreeMap<String,String>();
 "George"); f.put("George", "Martin");
 "John");
"Paul").equals("George")
"Dana").equals("John")
"Tom") == null
```


Map Views

```
ce Map<Key,Value> { // Continuation

Views of Maps */

of all keys. */
Set();

iset of all values that can be returned by get.
set is a collection that may have duplicates). */
alue> values();

of all(key, value) pairs */
y<Key,Value>> entrySet();
```


View Examples

rom a previous slide:

```
ing> f = new TreeMap<String,String>();
"George"); f.put("George", "Martin");
"John");
```

ious views of `f`:

```
String> i = f.keySet().iterator(); i.hasNext();)
==>  Dana, George, Paul
```

cinctly:

```
me : f.keySet())
 Dana, George, Paul

rent : f.values())
>   John, Martin, George

<String,String> pair : f.entrySet())
   (Dana,John), (George,Martin), (Paul,George)

ove("Dana");   // Now f.get("Dana") == null
```

Simple Banking I: Accounts

t a simple banking system. Can look up accounts by name
›sit or withdraw, print.

ure

```
 name, String number, int init) {
 name; this.number = number;
e = init;

lder's name */
ame;
mber */
umber;
lance */


 on STR in some useful format. */
ntStream str) { ... }
```


Simple Banking II: Banks

```
ples maintain mappings of String -> Account.  They keep
keys (Strings) in "compareTo" order, and the set of
ounts) is ordered according to the corresponding keys. */
ng,Account> accounts = new TreeMap<String,Account>();
ng,Account> names = new TreeMap<String,Account>();

nt(String name, int initBalance) {
 =
nt(name, chooseNumber(), initBalance);
t(acc.number, acc);
ame, acc);


tring number, int amount) {
= accounts.get(number);
ull) ERROR(...);
+= amount;

r withdraw.
```


Banks (continued): Iterating

count Data

```
l accounts sorted by number on STR. */
unt(PrintStream str) {
alues() is the set of mapped-to values.  Its
roduces elements in order of the corresponding keys.
account : accounts.values())
nt(str);


l bank accounts sorted by name on STR. */
(PrintStream str) {
account : names.values())
nt(str);
```

ion: What would be an appropriate representation for
d of all transactions (deposits and withdrawals) against

Partial Implementations

rfaces (like List) and concrete types (like LinkedList),
provides abstract classes such as AbstractList.

ke advantage of the fact that operations are related to

ce you know how to do get(k) and size() for an imple-
List, you can implement all the other methods needed
nly list (and its iterators).

n add(k,x) and you have all you need for the additional
f a growable list.

) and remove(k) and you can implement everything else.

The java.util.AbstractList helper class

```
t class AbstractList<Item> implements List<Item>

ted from List */
bstract int size();
bstract Item get(int k);
lean contains(Object x) {
 i = 0; i < size(); i += 1) {
 == null && get(i) == null) ||
 != null && x.equals(get(i))))
rn true;

false;

.: Throws exception; override to do more. */
t k, Item x) {
 UnsupportedOperationException();
```

remove, set

xample, continued: AListIterator

```
 abstract class AbstractList<Item>:
r<Item> iterator() { return listIterator(); }
rator<Item> listIterator() {
 AListIterator(this);


 class AListIterator implements ListIterator<Item> {
st<Item> myList;
or(AbstractList<Item> L) { myList = L; }
 position in our list. */
 0;

lean hasNext() { return where < myList.size(); }
 next() { where += 1; return myList.get(where-1); }
 add(Item x) { myList.add(where, x); where += 1; }
```

s, remove, set, etc.

Example: Using AbstractList

t to create a *reversed view* of an existing List (same
erse order).

```
 ReverseList<Item> extends AbstractList<Item> {
nal List<Item> L;

erseList(List<Item> L) { this.L = L; }

 size() { return L.size(); }

m get(int k) { return L.get(L.size()-k-1); }

d add(int k, Item x) { L.add(L.size()-k, x); }

m set(int k, Item x) { return L.set(L.size()-k-1, x); }

m remove(int k) { return L.remove(L.size() - k - 1); }
```


e: Another way to do AListIterator

e to make the nested class non-static:

```
r<Item> iterator() { return listIterator(); }
rator<Item> listIterator() { return this.new AListIterator(); }

AListIterator implements ListIterator<Item> {
position in our list. */
);

an hasNext() { return where < AbstractList.this.size(); }
next() { where += 1; return AbstractList.this.get(where-1); }
add(Item x) { AbstractList.this.add(where, x); where += 1; }
remove, set, etc.
```

actList.this means "the AbstractList I am attached
ew AListIterator means "create a new AListIterator
hed to X."

you can abbreviate this.new as new and can leave off
ctList.this parts, since meaning is unambiguous.

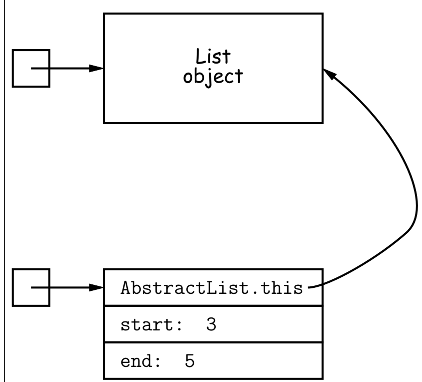
Getting a View: Sublists

blist(start, end) is a full-blown List that gives a
an existing list. Changes in one must affect the other.
rt of AbstractList:

```
ublist(int start, int end) {
 this.Sublist(start, end);


s Sublist extends AbstractList<Item> {
rror checks not shown
t start, end;
t start, int end) { obvious }

 size() { return end-start; }
m get(int k) { return AbstractList.this.get(start+k); }

d add(int k, Item x)
ctList.this.add(start+k, x); end += 1; }
```


What Does a Sublist Look Like?

= L.sublist(3, 5);



Arrays and Links

ys to represent a sequence: array and linked list

ary: ArrayList and Vector vs. LinkedList.

es: compact, fast ($\Theta(1)$) *random access* (indexing).

ages: insertion, deletion can be slow ($\Theta(N)$)

es: insertion, deletion fast once position found.

ages: space (link overhead), random access slow.

Implementing with Arrays

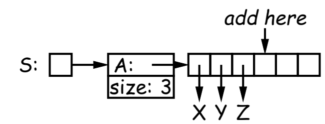
lem using arrays is insertion/deletion in the *middle* of a
ove things over).

ting from ends can be made fast:

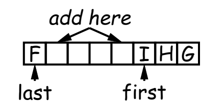
ray size to grow; amortized cost constant (Lecture #15).

one end really easy; classical stack implementation:

```
X");
Y");
Z");
```



rowth at either end, use *circular buffering:*



ccess still fast.

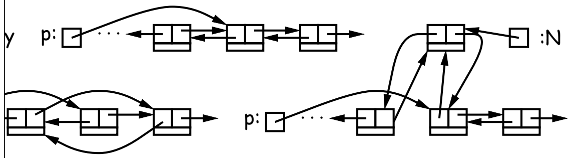
Clever trick: Sentinels

a dummy object containing no useful data except links.

ninate special cases and to provide a fixed object to
rder to access a data structure.

al cases ('**if**' statements) by ensuring that the first and
a list always have (non-null) nodes—possibly sentinels—
fter them:

```
list node at p:    // To add new node N before p:
 p.prev;           N.prev = p.prev; N.next = p;
 p.next;           p.prev.next = N;
                   p.prev = N;
```



Stacks and Recursion

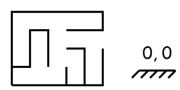
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

t(0)



Stacks and Recursion

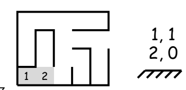
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

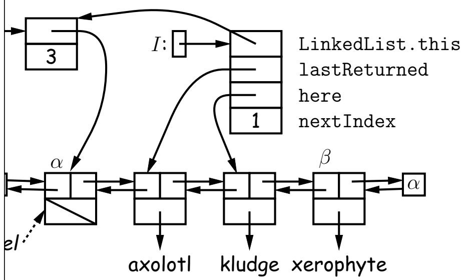
t(0)



Linking

f linking should now be familiar

a `LinkedList`. One possible representation for linked
erator object over it:




```
dList<String>();     I = L.listIterator();
l");                 I.next();
");
yte");
```


Specialization

pecial cases of general list:

ld and delete from one end (LIFO).

dd at end, delete from front (FIFO).

Add or delete at either end.

easily representable by either array (with circular buffer-
e or deque) or linked list.

e `List` types, which can act like any of these (although
ditional names for some of the operations).

va.util.Stack, a subtype of `List`, which gives tradi-
("push", "pop") to its operations. There is, however, no
face.

Stacks and Recursion

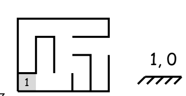
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

t(0)



Stacks and Recursion

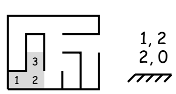
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

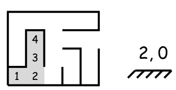
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

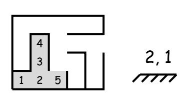
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

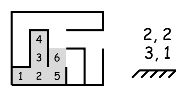
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

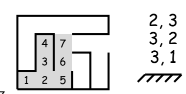
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

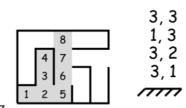
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```

t(0)



Stacks and Recursion

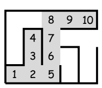
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



1, 3
3, 2
3, 1

Stacks and Recursion

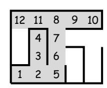
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



0, 2
3, 2
3, 1

Stacks and Recursion

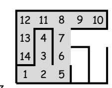
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



3, 2
3, 1

Stacks and Recursion

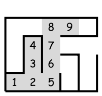
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



4, 3
1, 3
3, 2
3, 1

Stacks and Recursion

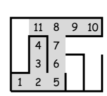
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



0, 3
3, 2
3, 1

Stacks and Recursion

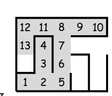
ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```
t):
tart)

sCrumb(start))
mb at start;
square, x,
t to start:
galPlace(x) && !isCrumb(x)
dExit(x)

t(0)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legalPlace(x) && !isCrumb(x)
            push x on S
```



0, 1
3, 2
3, 1

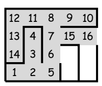
Stacks and Recursion

ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

t(0)



4, 3
3, 2
3, 1

oices: Extension, Delegation, Adaptation

d java.util.Stack type *extends* Vector:

```
tem> extends Vector<Item> { void push(Item x) { add(x); }
```

d have *delegated* to a field:

```
ack<Item> {
rayList<Item> repl = new ArrayList<Item>();
Item x) { repl.add(x); } ...
```

neralize, and define an *adapter:* a class used to make
e kind behave as another:

```
StackAdapter<Item> {
st repl;
k that uses REPL for its storage. */
ckAdapter(List<Item> repl) { this.repl = repl; }
d push(Item x) { repl.add(x); } ...


ack<Item> extends StackAdapter<Item> {
) { super(new ArrayList<Item>()); }
```

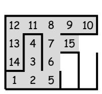

Stacks and Recursion

ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

t(0)



3, 3
3, 1

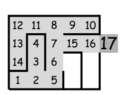
Stacks and Recursion

ed to *recursion*. In fact, can convert any recursive al-
tack-based (however, generally no great performance

me "push current variables and parameters, set param-
ew values, and loop."
comes "pop to restore variables and parameters."

```
t):                              findExit(start):
tart)                              S = new empty stack;
                                   push start on S;
sCrumb(start))                     while S not empty:
mb at start;                         pop S into start;
square, x,                           if isExit(start)
t to start:                            FOUND
galPlace(x) && !isCrumb(x)           else if (!isCrumb(start))
dExit(x)                               leave crumb at start;
                                       for each square, x,
                                         adjacent to start (in reverse):
                                           if legalPlace(x) && !isCrumb(x)
                                             push x on S
```

t(0)



3, 2
3, 1