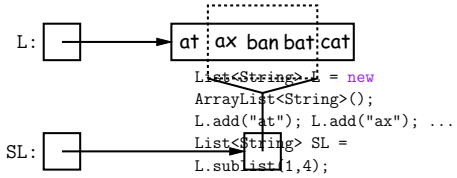
- Views
- Maps
- More partial implementations
- Array vs. linked: tradeoffs
- Sentinels
- Specialized sequences: stacks, queues, deques
- Circular buffering
- Recursion and stacks
- Adapters

sentation of (interface to) an existing object.

- For example, the `sublist` method is supposed to yield a "view of" part of an existing list:



- Example: after `L.set(2, "bag")`, value of `SL.get(1)` is `"bag"`, and after `SL.set(1,"bad")`, value of `L.get(2)` is `"bad"`.
- Example: after `SL.clear()`, `L` will contain only `"at"` and `"cat"`.
- Small challenge: "How do they *do* that?!"


```
package java.util;
public interface Map<Key,Value> {
  Value get(Object key);             // Value at
KEY.
  Object put(Key key, Value value);  // Set get(KEY)
-> VALUE
  ...
}
-------------------------------------------------------
Map<String,String> f = new TreeMap<String,String>();
f.put("Paul", "George"); f.put("George", "Martin");
f.put("Dana", "John");
// Now f.get("Paul").equals("George")
//     f.get("Dana").equals("John")
//     f.get("Tom") == null
```



```
            /* Views of Maps */

    /** The set of all keys. */
    Set<Key> keySet();

    /** The multiset of all values that can be returned
by get.
     *  (A multiset is a collection that may have
duplicates). */
    Collection<Value> values();

    /** The set of all(key, value) pairs */
    Set<Map.Entry<Key,Value>> entrySet();
  }
```



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

we can take various views of `f`:

```
for (Iterator<String> i = f.keySet().iterator();
i.hasNext();)
    i.next()  ===>  Dana, George, Paul
// or, more succinctly:
for (String name : f.keySet())
    name  ===>  Dana, George, Paul

for (String parent : f.values())
    parent  ===>   John, Martin, George

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

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


look up accounts by name or number, deposit or withdraw, print.

Account Structure

```
class Account {
  Account(String name, String number, int init) {
    this.name = name; this.number = number;
    this.balance = init;
  }
  /** Account-holder's name */
  final String name;
  /** Account number */
  final String number;
  /** Current balance */
  int balance;

  /** Print THIS on STR in some useful format. */
  void print(PrintStream str) { ... }
}
```

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

  void openAccount(String name, int initBalance) {
     Account acc =
       new Account(name, chooseNumber(), initBalance);
     accounts.put(acc.number, acc);
     names.put(name, acc);
  }

  void deposit(String number, int amount) {
    Account acc = accounts.get(number);
    if (acc == null) ERROR(...);
    acc.balance += amount;
  }
  // Likewise for withdraw.
```



```
/** Print out all accounts sorted by number on STR.
*/
void printByAccount(PrintStream str) {
   // accounts.values() is the set of mapped-to values.
Its
   // iterator produces elements in order of the corresponding
keys.
   for (Account account : accounts.values())
     account.print(str);
}

/** Print out all bank accounts sorted by name on
STR. */
void printByName(PrintStream str) {
   for (Account account : names.values())
     account.print(str);
}
```

A Design Question: What would be an appropriate representation for keeping a record of all transactions (deposits and withdrawals)

types (like `LinkedList`), Java library provides abstract classes such as `AbstractList`.

- Idea is to take advantage of the fact that operations are related to each other.
- Example: once you know how to do `get(k)` and `size()` for an implementation of `List`, you can implement all the other methods needed for a *read-only* list (and its iterators).
- Now throw in `add(k,x)` and you have all you need for the additional operations of a growable list.
- Add `set(k,x)` and `remove(k)` and you can implement everything else.


```
public abstract class AbstractList<Item> implements
List<Item> {
   /** Inherited from List */
   // public abstract int size();
   // public abstract Item get(int k);
   public boolean contains(Object x) {
      for (int i = 0; i < size(); i += 1)
{
        if ((x == null && get(i) == null)
||
            (x != null && x.equals(get(i))))
          return true;
      }
      return false;
   }
   /* OPTIONAL: Throws exception; override
to do more. */
   void add(int k, Item x) {
     throw new UnsupportedOperationException();
```

```
AbstractList<Item>:
public Iterator<Item> iterator() { return
listIterator(); }
public ListIterator<Item> listIterator() {
   return new AListIterator(this);
}

private static class AListIterator
implements ListIterator<Item> {
   AbstractList<Item> myList;
   AListIterator(AbstractList<Item> L) {
myList = L; }
   /** Current position in our list. */
   int where = 0;

   public boolean hasNext() { return where <
myList.size(); }
   public Item next() { where += 1; return
myList.get(where-1); }
   public void add(Item x) {
myList.add(where, x); where += 1; }
   ... previous, remove, set, etc.
}
...
```


It's also possible to make the nested class non-static:

```
public Iterator<Item> iterator() { return
listIterator(); }
public ListIterator<Item> listIterator() { return
this.new AListIterator(); }

private class AListIterator implements
ListIterator<Item> {
  /** Current position in our list. */
  int where = 0;

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


attached to `A`.

- In this case you can abbreviate `this.new` as `new` and can leave off some `AbstractList.this` parts, since meaning is unambiguous.

an existing `List` (same elements in reverse order). Operations on the original list affect the view, and vice-versa.


```
  public ReverseList(List<Item> L) { this.L
= L; }

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

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

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

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

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

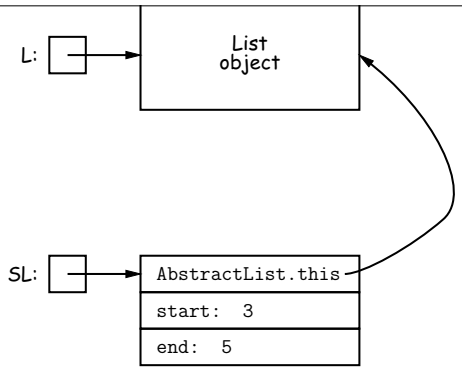
gives a view of part of an existing list. Changes in one must affect the other. How?


```
    return this.new Sublist(start, end);
  }

  private class Sublist extends
  AbstractList<Item> {
    private int start, end;
    Sublist(int start, int end) { obvious }

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

    public void add(int k, Item x)
      { AbstractList.this.add(start+k, x); end
  += 1; }
    ...
  }
```

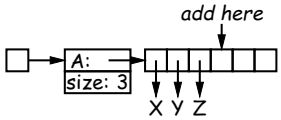
ray and linked list

- In Java Library: `ArrayList` and `Vector` vs. `LinkedList`.
- Array:
  - Advantages: compact, fast ($\Theta(1)$) *random access* (indexing).
  - Disadvantages: insertion, deletion can be slow ($\Theta(N)$)
- Linked list:
  - Advantages: insertion, deletion fast once position found.
  - Disadvantages: space (link overhead), random access slow.

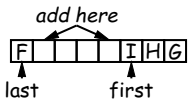
in the *middle* of a list (must shove things over).

- Adding/deleting from ends can be made fast:
  - Double array size to grow; amortized cost constant (Lecture #15).
  - Growth at one end really easy; classical stack implementation:

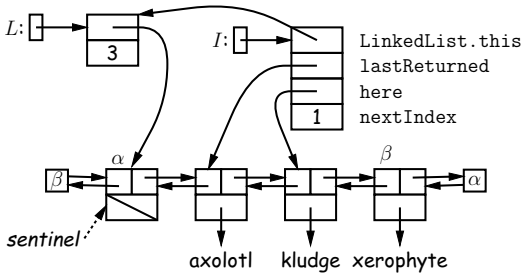
```
S.push("X");
S.push("Y");
S.push("Z");
```



  - To allow growth at either end, use *circular buffering:*



- Used in Java `LinkedList`. One possible representation for linked list and an iterator object over it:




```
L = new LinkedList<String>();
L.add("axolotl");
L.add("kludge");
L.add("xerophyte");
```

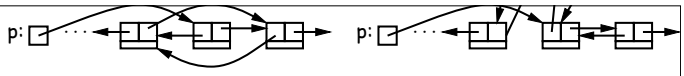
```
I = L.listIterator();
I.next();
```


useful data except links.

- Used to eliminate special cases and to provide a fixed object to point to in order to access a data structure.
- Avoids special cases ('**if**' statements) by ensuring that the first and last item of a list always have (non-null) nodes—possibly sentinels—before and after them:
-

```
// To delete list node at p:
p.next.prev = p.prev;
p.prev.next = p.next;
```

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

  - **Stack**: Add and delete from one end (LIFO).
  - **Queue**: Add at end, delete from front (FIFO).
  - **Dequeue**: Add or delete at either end.
- All of these easily representable by either array (with circular buffering for queue or deque) or linked list.
- Java has the `List` types, which can act like any of these (although with non-traditional names for some of the operations).
- Also has `java.util.Stack`, a subtype of `List`, which gives traditional names ("push", "pop") to its operations. There is, however, no "stack" interface.

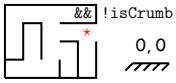
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
&& !isCrumb(x)
  push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



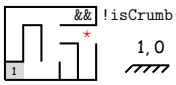
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
        push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



1, 0

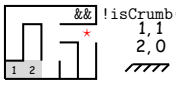
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



1, 1
2, 0

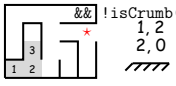
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



1, 2
2, 0

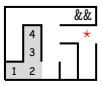
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
&& !isCrumb(x)
    push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



2, 0

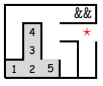
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
&& !isCrumb(x)
    push x on S

Call: findExit((0,0))
Exit: (4, 2)



2, 1

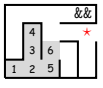
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
&& !isCrumb(x)
    push x on S

Call: findExit((0,0))
Exit: (4, 2)



2, 2
3, 1

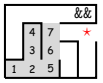
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
&& !isCrumb(x)
        2, 3
        3, 2
        3, 1
        push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



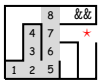
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
&& !isCrumb(x)
        3, 3
        2, 3
        3, 2
        3, 1
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



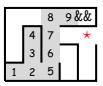
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
&& !isCrumb(x)
        4, 3
        2, 3
        3, 2
        3, 1
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



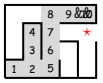
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
!isCrumb(x)
        push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



1, 3
3, 2
3, 1

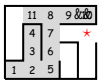
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
!isCrumb(x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



0, 3
3, 2
3, 1

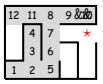
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
!isCrumb(x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



0, 2
3, 2
3, 1

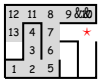
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
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 legal(start,x)
        push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



!isCrumb(x)
0, 1
3, 2
3, 1

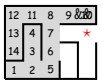
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



!isCrumb(x)
3, 2
3, 1

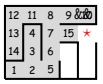
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."

(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)

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 legal(start,x)
        push x on S

Call: findExit((0,0))
Exit: (4, 2)



!isCrumb(x)
4, 2
3, 1

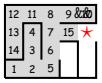
convert any recursive algorithm to stack-based (however, generally no great performance benefit):

- Calls become "push current variables and parameters, set parameters to new values, and loop."
- Return becomes "pop to restore variables and parameters."


```
(!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x)
&& !isCrumb(x)
          findExit(x)
```

```
    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 legal(start,x)
!isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



- The standard `java.util.Stack` type *extends* `Vector`:

```
class Stack<Item> extends Vector<Item> { void push(Item
x) { add(x); } ... }
```

- Could instead have *delegated* to a field:

```
class ArrayStack<Item> {
   private ArrayList<Item> repl = new ArrayList<Item>();
   void push(Item x) { repl.add(x); } ...
}
```

- Or, could generalize, and define an *adapter:* a class used to make objects of one kind behave as another:

```
public class StackAdapter<Item> {
   private List repl;
   /** A stack that uses REPL for its storage.
*/
   public StackAdapter(List<Item> repl) { this.repl
= repl; }
   public void push(Item x) { repl.add(x); } ...
```



```
   ArrayStack() { super(new ArrayList<Item>());
}
}
```