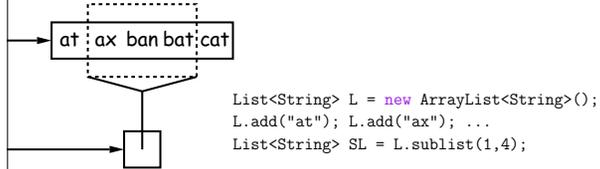


## Views

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

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



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

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

Quote: "How do they do that?!"

## Map Views

```
interface Map<Key,Value> { // Continuation
    // Views of Maps */
    // of all keys. */
    Set<Key> keySet();

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

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

## Simple Banking I: Accounts

Implement a simple banking system. Can look up accounts by name, deposit or withdraw, print.

Structure

```
Account(String name, String number, int init) {
    name; this.number = number;
    balance = init;

    // Leader's name */
    name;
    // number */
    number;
    // balance */
    balance;

    // Print on STR in some useful format. */
    toString(String str) { ... }
}
```

## CS61B Lecture #18: Assorted Topics

Implementations

Tradeoffs

Sequences: stacks, queues, deques

Priority queues

Linked lists

## Maps

Definition of "modifiable function:"

```
interface Map<Key,Value> {
    Value get(Object key); // Value at KEY.
    Value put(Key key, Value value); // Set get(KEY) -> VALUE
}
```

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

## View Examples

Example from a previous slide:

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

Various views of f:

```
f.keySet().iterator().hasNext()
// => Dana, George, Paul
// cinctly:
f.keySet()
// Dana, George, Paul

f.values()
// John, Martin, George

f.entrySet()
// (Dana,John), (George,Martin), (Paul,George)

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

## Banks (continued): Iterating

### Account Data

```
Print accounts sorted by number on STR. */
void print(PrintStream str) {
    // values() is the set of mapped-to values. Its
    // iterator produces elements in order of the corresponding keys.
    Iterator<Account> accounts = accounts.values().iterator();
    while (accounts.hasNext())
        print(str, accounts.next());
}
```

```
Print bank accounts sorted by name on STR. */
void print(PrintStream str) {
    Iterator<Account> accounts = names.values().iterator();
    while (accounts.hasNext())
        print(str, accounts.next());
}
```

**Question:** What would be an appropriate representation for the set of all transactions (deposits and withdrawals) against a given account?

4:22 2021

CS61B: Lecture #18 8

## The java.util.AbstractList helper class

```
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 (get(i) == null && x == null) ||
                (get(i) != null && x.equals(get(i)))
                return true;
        }
        return false;
    }
    // Throws exception; override to do more. */
    public void add(int k, Item x) {
        throw new UnsupportedOperationException();
    }
    // remove, set
}
```

4:22 2021

CS61B: Lecture #18 10

## Example: Another way to do AListIterator

Example to make the nested class non-static:

```
public class AbstractList<Item> {
    private ListIterator<Item> listIterator() { return new AListIterator(); }
    private ListIterator<Item> listIterator(int where) { return this.new AListIterator(where); }

    private class AListIterator implements ListIterator<Item> {
        private int where; // position in our list. */
        AListIterator() { where = 0; }
        AListIterator(int where) { where = where; }
        boolean hasNext() { return where < AbstractList.this.size(); }
        Item next() { where += 1; return AbstractList.this.get(where-1); }
        void add(Item x) { AbstractList.this.add(where, x); where += 1; }
        // remove, set, etc.
    }
}
```

`AbstractList.this` means "the AbstractList I am attached to".  
`new AListIterator` means "create a new AListIterator attached to X."

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

4:22 2021

CS61B: Lecture #18 12

## Simple Banking II: Banks

Banks maintain mappings of String -> Account. They keep keys (Strings) in "compareTo" order, and the set of values (Accounts) is ordered according to the corresponding keys. \*/

```
Map<String,Account> accounts = new TreeMap<String,Account>();
Map<String,Account> names = new TreeMap<String,Account>();

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

```
void withdraw(String number, int amount) {
    Account acc = accounts.get(number);
    if (acc == null) ERROR(...);
    acc.withdraw(amount);
}
```

void withdraw.

4:22 2021

CS61B: Lecture #18 7

## Partial Implementations

Interfaces (like List) and concrete types (like LinkedList), provide abstract classes such as AbstractList.

One advantage of the fact that operations are related to interfaces is that you can implement only a few methods.

For example, if you know how to do `get(k)` and `size()` for an implementation of List, you can implement all the other methods needed for a growable list (and its iterators).

For example, if you know how to do `add(k,x)` and you have all you need for the additional methods needed for a growable list.

For example, if you know how to do `add(k,x)` and `remove(k)` and you can implement everything else.

4:22 2021

CS61B: Lecture #18 9

## Example, continued: AListIterator

```
public abstract class AbstractList<Item> {
    private ListIterator<Item> listIterator() { return new AListIterator(); }
    private ListIterator<Item> listIterator(int where) { return this.new AListIterator(where); }

    private class AListIterator implements ListIterator<Item> {
        private List<Item> myList;
        AListIterator(AbstractList<Item> L) { myList = L; }
        private int where; // position in our list. */
        AListIterator() { where = 0; }
        AListIterator(int where) { where = where; }
        boolean hasNext() { return where < myList.size(); }
        Item next() { where += 1; return myList.get(where-1); }
        void add(Item x) { myList.add(where, x); where += 1; }
        // remove, set, etc.
    }
}
```

4:22 2021

CS61B: Lecture #18 11



## Specialization

Special cases of general list:

add and delete from one end (LIFO).

add at end, delete from front (FIFO).

Add or delete at either end.

Not easily representable by either array (with circular buffer or deque) or linked list.

Use List types, which can act like any of these (although with additional names for some of the operations).

java.util.Stack, a subtype of List, which gives traditional ("push", "pop") to its operations. There is, however, no face.

4:22 2021

CS61B: Lecture #18 20

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

we "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```

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



4:22 2021

CS61B: Lecture #18 22

## Stacks and Recursion

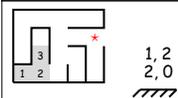
related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

we "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```

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



4:22 2021

CS61B: Lecture #18 24

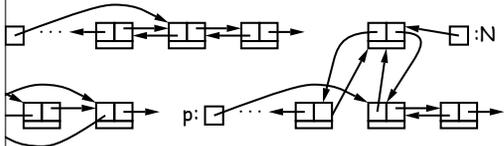
## Clever trick: Sentinels

Use a dummy object containing no useful data except links. Eliminate special cases and to provide a fixed object to refer to access a data structure.

Handle special cases ('if' statements) by ensuring that the first and last nodes in a list always have (non-null) nodes—possibly sentinels—before and after them:

```

listNode 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;
    
```



24:22 2021

CS61B: Lecture #18 19

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

we "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```

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



4:22 2021

CS61B: Lecture #18 21

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

we "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```

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



4:22 2021

CS61B: Lecture #18 23

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

2,1

4:22 2021

CS61B: Lecture #18 26

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

2,3  
3,2  
3,1

4:22 2021

CS61B: Lecture #18 28

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

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

4:22 2021

CS61B: Lecture #18 30

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

2,0

4:22 2021

CS61B: Lecture #18 25

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

2,2  
3,1

4:22 2021

CS61B: Lecture #18 27

## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

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

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

4:22 2021

CS61B: Lecture #18 29

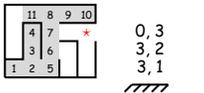
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



0,3  
3,2  
3,1

4:22 2021

CS61B: Lecture #18 32

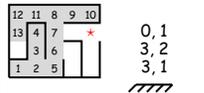
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



0,1  
3,2  
3,1

4:22 2021

CS61B: Lecture #18 34

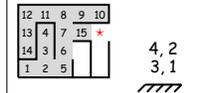
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



4,2  
3,1

4:22 2021

CS61B: Lecture #18 36

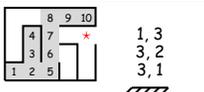
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



1,3  
3,2  
3,1

4:22 2021

CS61B: Lecture #18 31

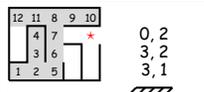
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



0,2  
3,2  
3,1

4:22 2021

CS61B: Lecture #18 33

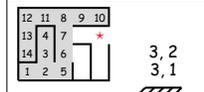
## Stacks and Recursion

related to recursion. In fact, can convert any recursive stack-based (however, generally with no great performance):

me "push current variables and parameters, set parameter values, and loop."

comes "pop to restore variables and parameters."

```
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 legal(start,x) && !isCrumb(x)
                    push x on S
mb(start))
t start;
re, x,
start:
start,x) && !isCrumb(x)
t(x)
```



3,2  
3,1

4:22 2021

CS61B: Lecture #18 35

