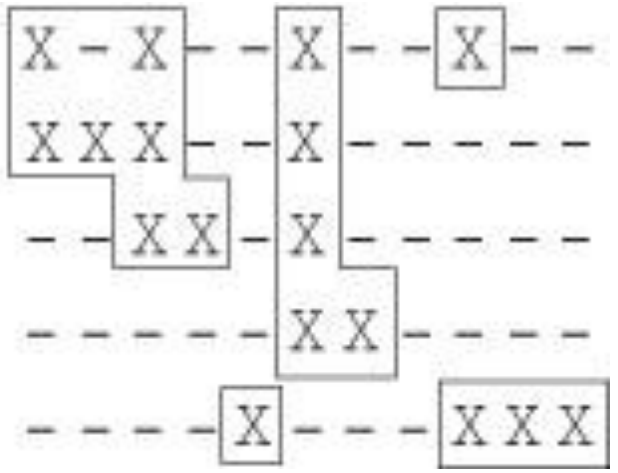
4.4 Counting Blobs

- A blob is a contiguous collection of X's.
- Two X's in a grid are considered contiguous if they are beside each other horizontally or vertically. For example:

```
X - X - - X - - X - -
X X X - - X - - - - -
- - X X - X - - - - -
- - - - - X X - - - -
- - - - X - - - X X X
```



Generating Blob Grids based on a Percentage

```
for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++)
  {
    randInt = rand.nextInt(100);  // random number 0 .. 99
    if (randInt < percentage)
      grid[i][j] = true;
    else
      grid[i][j] = false;
  }
```

For example:

```
-----------      -X--X----X-      X-X-XXX-XX-       XXXXXXXXXXX
-----------      X----X-X---      XXXXX-XXXXX       XXXXXXXXXXX
-----------      ---X-----XX      XXXXX---XX-       XXXXXXXXXXX
-----------      XX-X-X--X--      ---X---XXX-       XXXXXXXXXXX
-----------      ------XX—X       XX--X-XXXXX       XXXXXXXXXXX

Percentage 0     Percentage 33    Percentage 67    Percentage 100
```

Incorrect Counting Algorithm

```
int count = 0;
for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++)
    if (grid[i][j])
      count++;
```

Problem this code counts the number of blob characters, not the number of blobs.

Whenever we encounter a blob character and count it, we must somehow mark all of the characters within that blob as having been counted.

To support this approach we create a "parallel" grid of boolean called visited.

Correct Counting Algorithm

We need a method markBlob, that accepts as arguments the row and column indexes of a blob character, and proceeds to "mark" all the characters within that blob as having been visited.

```
int count = 0;
for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++)
    if (grid[i][j] && !visited[i][j])
    {
      count++;
      markBlob(i, j);
    }
```

The Marking Algorithm

- The markBlob method is passed the location of a blob character that needs to be marked, through its two arguments, row and col. It does that:

```
visited[row][col] = true;
```

- It must also mark all of the other characters in the blob. It checks the four locations "around" that location, to see if they need to be marked.

- When should one of those locations be marked? There are three necessary conditions:
  - the location exists, that is, it is not outside the boundaries of the grid
  - the location contains a blob character
  - the location has not already been visited

The Marking Algorithm

- For example, the following code checks and marks the location above the indicated location (note the recursion):

```
if ((row - 1) >= 0)            // if its on the grid
  if (grid[row - 1][col])        // and has a blob character
    if (!visited[row - 1][col])   // and has not been visited
      markBlob(row - 1, col);       // then mark it
```

- The code for the remaining three directions (down, left, and right) is similar.

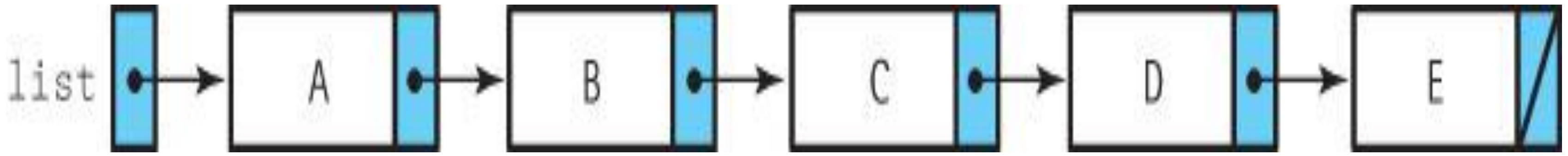
Three Question Approach

- Base-Case
  - Is there a nonrecursive way out?
  - Does the method work for the base-case?
  - Yes: mark location as visited
- Smaller-Case
  - Does each recursive call involve a smaller case that leads to the base case.
  - Yes: each call → a blob is marked visited
- General-Case
  - Assume recursive calls work correctly, does the method work in the general case?
  - Yes: each location in the blob connects to another, all locations will be checked and marked

Code and Demo

- Walk through the code contained in Grid.java and BlobApp.java, and demonstrate the running program.

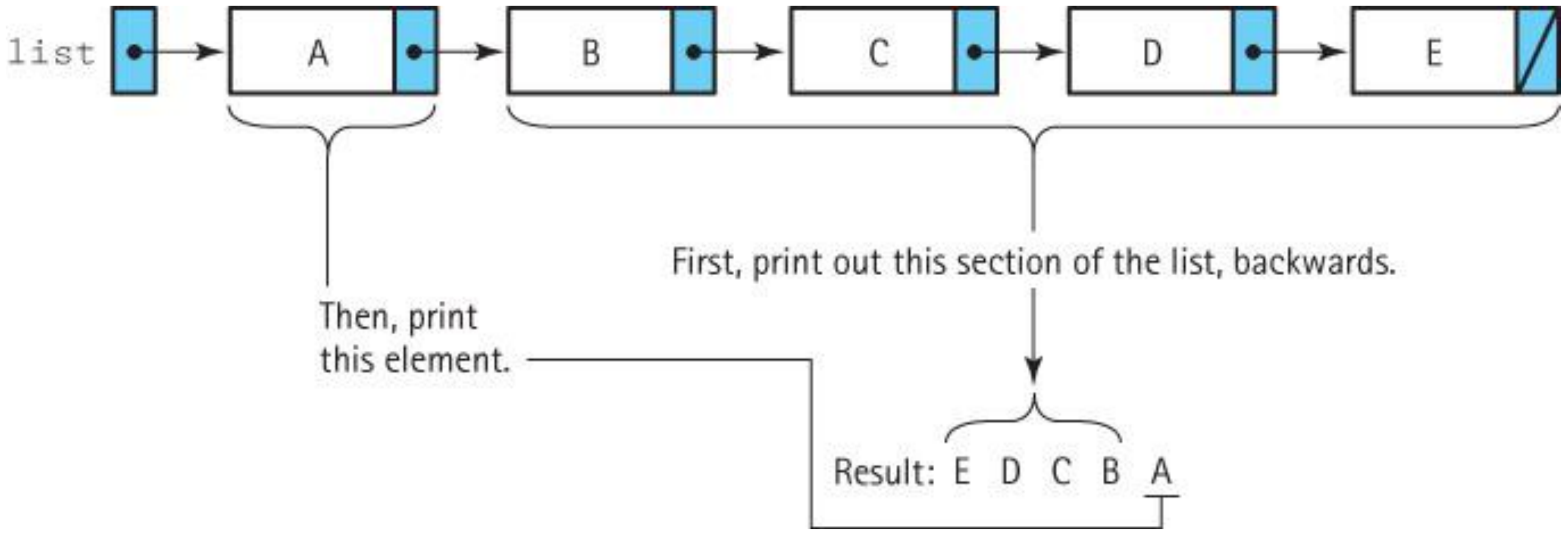
4.5 Recursive Linked-List Processing



- Reverse Printing: Our goal is to print the elements of a linked list in *reverse* order.

- This problem is much more easily and elegantly solved recursively than it is iteratively.

Recursive Reverse Print


list
A
B
C
D
E
First, print out this section of the list, backwards.
Then, print this element.
Result: E D C B A

Recursive revPrint (listRef)

Print out the second through last elements in the list referenced by listRef in reverse order.

Then print the first element in the list referenced by listRef

Extending LinkedStack with a Reverse Print

```
import ch03.stacks.*;
import support.LLObjectNode;

public class LinkedStack2<T> extends LinkedStack<T>
{
  private void revPrint(LLNode<T> listRef)
  {
    if (listRef != null)
    {
      revPrint(listRef.getLink());
      System.out.println(" " + listRef.getInfo());
    }
  }

  public void printReversed()
  {
    revPrint(top);
  }
}
```

4.6 Removing Recursion

- Two substitutes for recursion
  - iteration
  - stacking

- First, examine how recursion is implemented
  - Understanding how recursion works helps us see how to develop non-recursive solutions.

Static Storage Allocation

- A compiler that translates a high-level language program into machine code for execution on a computer must
  - reserve space for the program variables
  - translate the high level executable statements into equivalent machine language statements

Example of Static Allocation

Consider the following program:

```
public class Kids
{
  private static int countKids(int girlCount, int boyCount)
  {
    int totalKids;
    . . .
  }

  public static void main(String[] args)
  {
    int numGirls; int numBoys; int numChildren;
    . . .
  }
}
```

A compiler could create two separate machine code units for this program, one for the countKids method and one for the main method. Each unit would include space for its variables plus the sequence of machine language statements that implement its high-level code.

Limitations of static allocation

- Static allocation like this is the simplest approach possible. But it does not support recursion.

- The space for the countKids method is assigned to it at compile time. This works fine when the method will be called once and then always return before it is called again. But a recursive method can be called again and again before it returns. Where do the second and subsequent calls find space for their parameters and local variables?

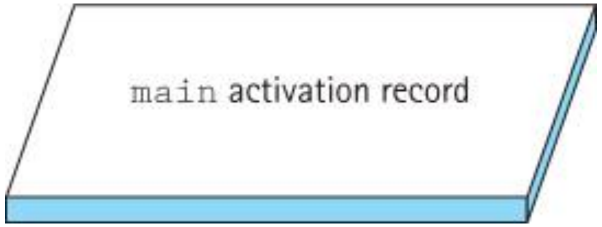
- Therefore dynamic storage allocation is needed.

Dynamic Storage Allocation

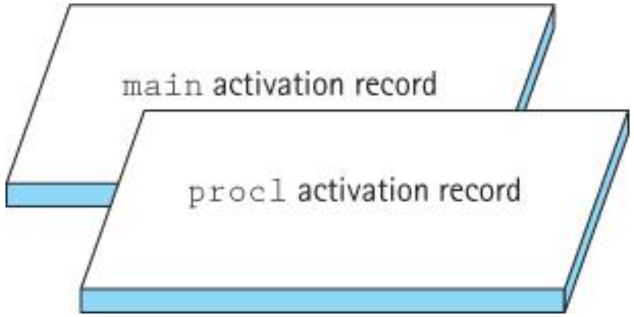
- Dynamic storage allocation
  - Provides space for a method when it is called
- When a method is invoked
  - Space is required to hold its parameters
    - local variables
    - return address
      - the address in the calling code to which the computer returns when the method completes its execution
- This space is called
  - **activation record** or **stack frame**

Dynamic Storage Allocation

Consider a program whose main method calls proc1, which then calls proc2. When the program begins executing, the “main” activation record is generated:

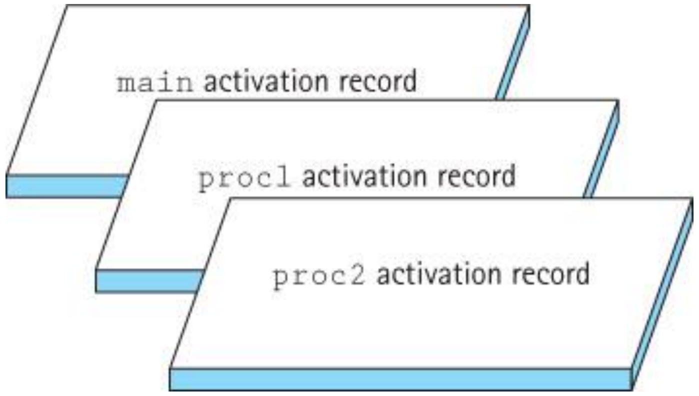


At the first method call, an activation record is generated for proc1:

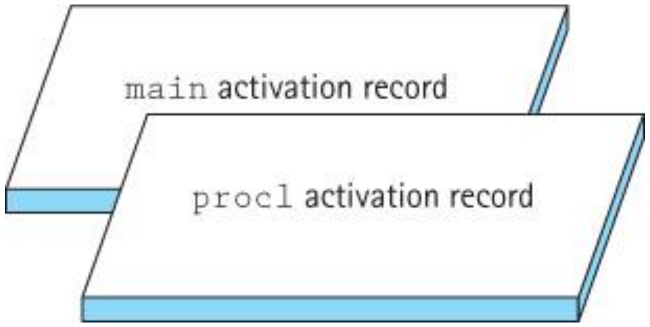


Dynamic Storage Allocation

When proc2 is called from within proc1, its activation record is generated. Because proc1 has not finished executing, its activation record is still around:



When proc2 finishes executing, its activation record is released:



Dynamic Storage Allocation

How is memory handled when a program runs?

- The order of activation follows the Last-In-First-Out rule.
- **Run-time or system stack:** A system data structure that keeps track of activation records during the execution of a program
- Each nested level of method invocation adds another activation record to the stack.
- As each method completes its execution, its activation record is popped from the stack.
- Recursive method calls, like calls to any other method, cause a new activation record to be generated.
- **Depth of recursion :** The number of activation records on the system stack, associated with a given a recursive method

Removing Recursion - Iteration

- Suppose the recursive call is the last action executed in a recursive method (tail recursion)
  - The recursive call causes an activation record to be put on the run-time stack that contains the invoked method's arguments and local variables.
  - When this recursive call finishes executing, the run-time stack is popped and the previous values of the variables are restored.
  - Execution continues where it left off before the recursive call was made.
  - But, because the recursive call is the last statement in the method, there is nothing more to execute and the method terminates without using the restored local variable values.
- In such a case the recursion can be replaced with iteration.

Example of eliminating tail recursion

```
public static int factorial(int n)
{
  if (n == 0)
    return 1;          // Base case
  else
    return (n * factorial(n - 1));     // General case
}
```

Declare a variable to hold the intermediate values;

initialize it to the value returned in the base case.

Use a *while* loop so that each time through the loop corresponds to one recursive call.

The loop should continue processing until the base case is met:

```
private static int factorial(int n)
{
  int retValue = 1;   // return value
  while (n != 0)
  {
    retValue = retValue * n;
    n = n - 1;
  }
  return(retValue);
}
```

Remove Recursion - Stacking

- When the recursive call is not the last action executed in a recursive method, we cannot simply substitute a loop for the recursion.

- In such cases we can “mimic” recursion by using our own stack to save the required information when needed, as shown in the Reverse Print example on the next slide.

4.7 When to use recursion?

- Main issues
  - Efficiency of the solution
  - Clarity of the solution

- Demo LinkedStack
  - Recursive: UseStack2
  - Non-Recursive: UseStack3

Efficiency Considerations

- Recursion Overhead
  - Greater "overhead" than a non-recursive solutions
    - An increased number of method calls leads to increased
    - Processing time:
      - create and dispose of the activation record
      - manage the run-time stack
    - Space (memory requirements)
      - activation records must be stored
- Inefficient Algorithms
  - Another potential problem is that a particular recursive solution might be inherently inefficient.
    - Repeatedly solving the same sub-problem

Clarity

- Recursive solutions are often simpler and more natural for the programmer to write.
  - The work provided by the system stack is “hidden” and therefore a solution may be easier to understand.
- Example: printing a linked list in reverse order
  - Recursive version: system handles stacking
  - Non-recursive version: we explicitly handle stacking

HW5 is now a lab!

- Open StackBubbleSort.zip
- Pseudocode scrambleArray is now coded (you're welcome!)
- So what is bubbleSort… Other sorting routines
  - Check out wikipedia, search on "sorting algorithms"
- Revisit:
  - ArrayStack, StackInterface, BoundedStackInterface, StackOverflowException, StackUndeflowException

Homework 5

- Book questions CH4: 1, 2, 3, 6, 7
- Extra credit
  - Search wikipedia for “sorting algorithms”
  - Implement quicksort (5 points)
    - (1 point) Implement the scrambleArray method
    - (2 points) Benchmarking
      - Random values range from 1 – 1000
      - Array sizes: 10, 100, 1000, 10000, 100000
    - (2 points) Create a test driver
      - implement benchmarking