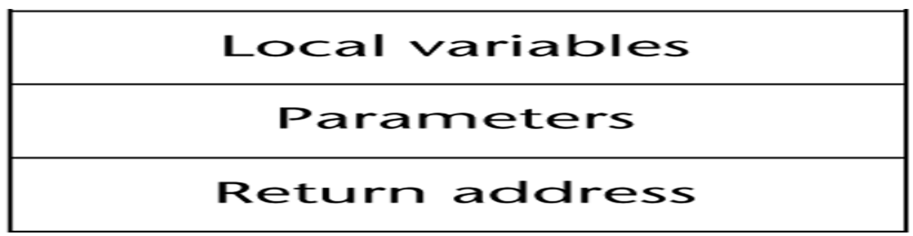
Activation Records

The storage (for formals, local variables, function results etc.) needed for execution of a subprogram is organized as an activation record.

An Activation Record for “Simple” Subprograms

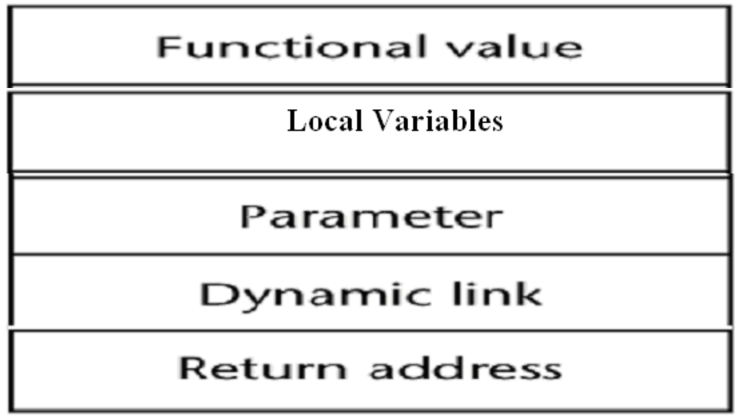


.

Activation Record for a Language with Stack-Dynamic Local Variables

Dynamic link: points to the top of an activation record of the caller

Activation Record for Recursion

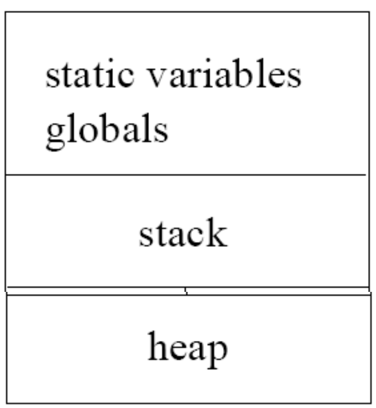


Allocation of activation records can be:
• on the heap
– used in Modula-3, LISP, Scheme, ...
• on the stack
– used in C, C++, Java, C#, Pascal, Ada, ...

Subprogram in C

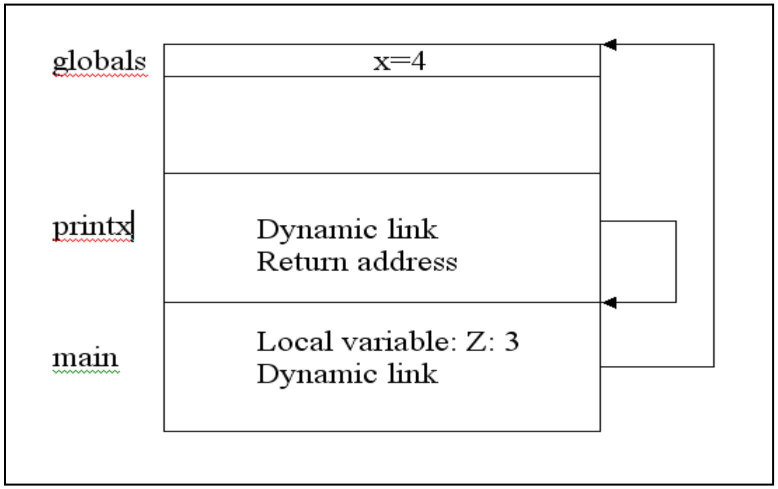
- the lifetime of local variables is contained within one activation (except for static variables)
- locals can be allocated at activation time and deallocated when the activation ends
- activation records can be allocated on a stack

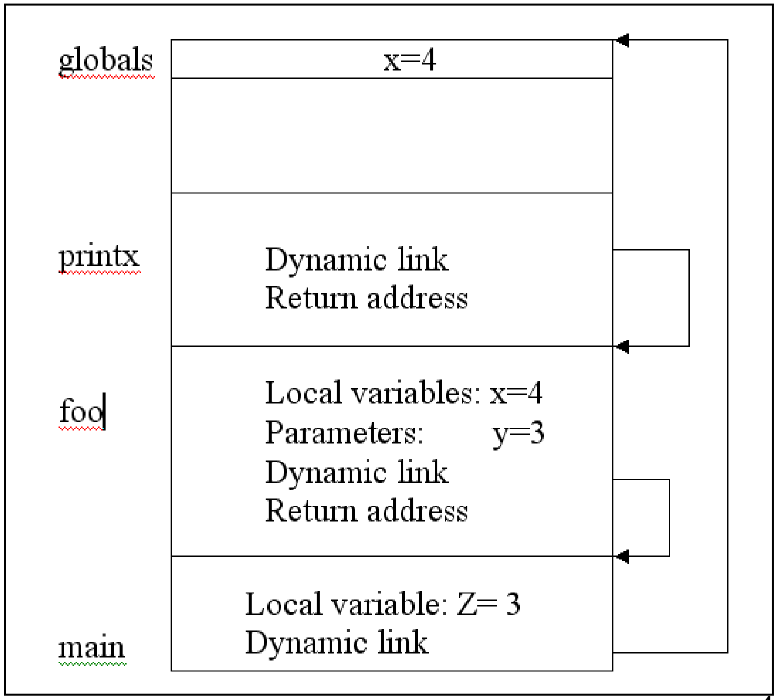
Memory layout for C programs:



Shows the Activation Record during the first and second execution of printx():

```
#include <stdio.h>
int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
int x = 4;
x = x + x * y;
printx();
}

void main() {
int z = 3;
printx();
foo(z);
}
```





Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, JavaScript) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance of enclosing scopes in the stack
- A reference to a non-locally variable in a static-scoped language with nested subprograms requires a two step access process:
    1. Find the correct activation record instance
    2. Determine the correct offset within that activation record instance

Static Scoping of Nested subprograms

- In this approach, a new pointer, called a static link, is added to the activation record.

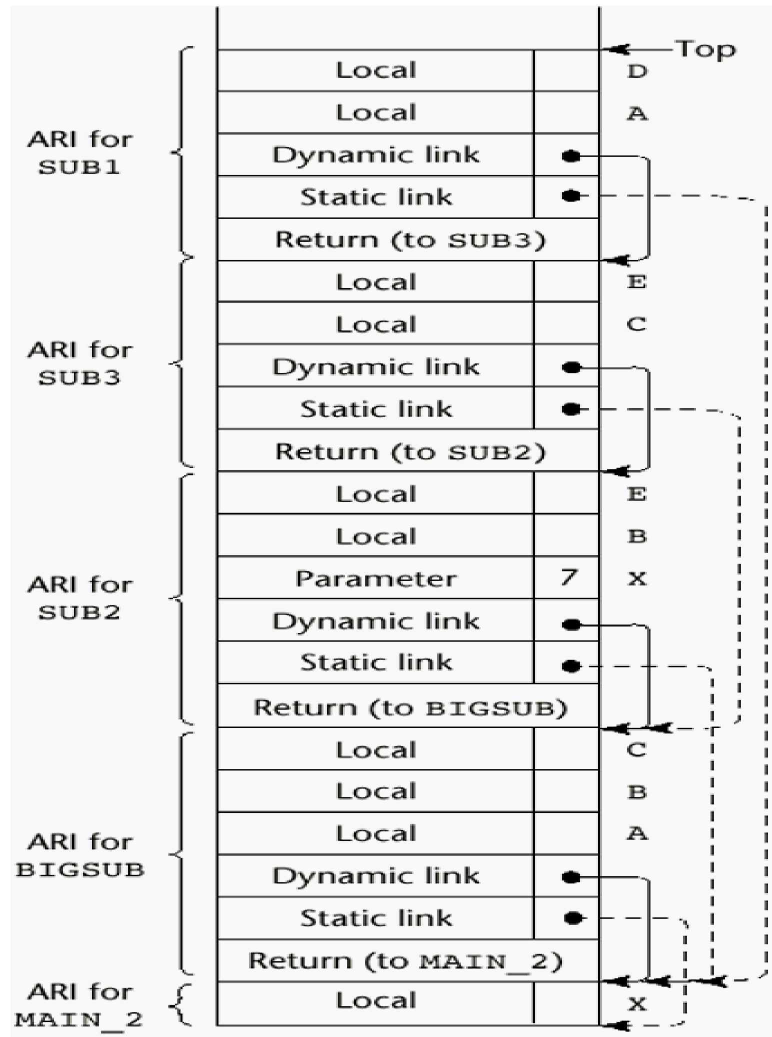
Local Variables
Parameter
Dynamic link
Static link
Return address

- The static link in an activation record instance for subprogram A points to the bottom of the activation record instances of A's static parent

```
program MAIN_2; //Example Pascal Program
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <----------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <----------------3
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end; { MAIN_2 }
```

- Call sequence for **MAIN_2 ?**

Activation Records at Position 1



- A *static chain* is a chain of static links that connects certain activation record instances
- The static chain from an activation record instance connects it to all of its static ancestors

Sub3 → Sub2 → BigSub → Main2

Finding the correct activity record instance of a nonlocal variable using static links is relatively straightforward.

Point 1:

To access nonlocal variable B? C?

After Sub1 complete its execution, the activation record instance for Sub1 is removed from the stack, and control return to Sub3.

Point 2: to access E? B? A?

Blocks

• **Blocks are user-specified local scopes for variables.** It is legal to declare variables within blocks contained within other blocks.

• An example in C

```
void SquareTable(int lower, int upper){
 int n;
 for (n = lower; n <= upper; n++)  {
  int square;
  square = n * n;
  printf("%8d%8d\n", n, square);  }
 }
```

(a) When does square get allocated and deallocated?

The memory is allocated and deallocated on each pass through the inner block.

(b) How should the memory diagram be drawn?

Implementing Blocks

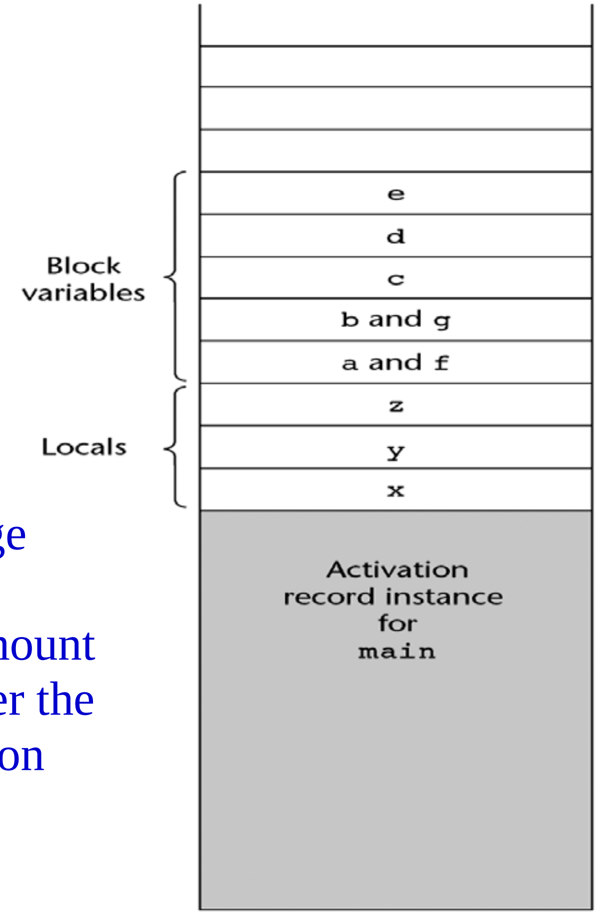
1.Treat blocks as parameter-less subprograms that are always called from the same location.

– Every block has an activation record; an instance is created every time the block is executed

```
void main(){
    int x, y, z;
   while ( … ){
     int a, b, c;
     ……
     while ( …){
         int d, e;
           …… }
   }
  while ( … ) {
      int f, g;
       ……  }  ……}
```

Figure 10.10

Block variable storage when blocks are not treated as parameterless procedures



2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

One way that local variables and non-local references can be implemented in a dynamic-scoped language:

- *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain

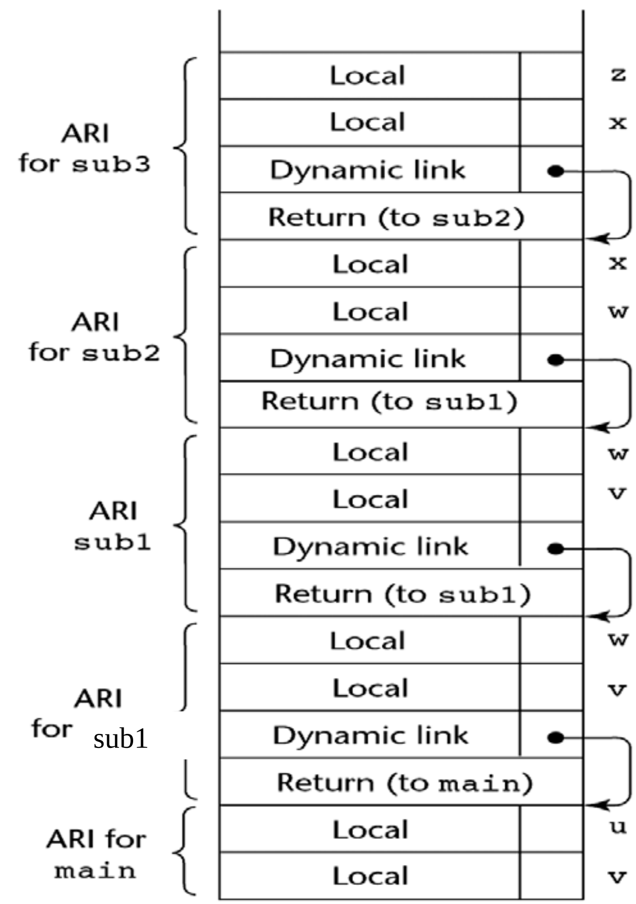
```
void sub3(){
  int x, y;
  x = u+v;}

void sub2(){
  int w, x;}

void sub1(){
  int v, w;}

void main(){
  int v, u;}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

Access u in sub3?
Access v in sub3?
Access x in sub3?

Figure 10.11

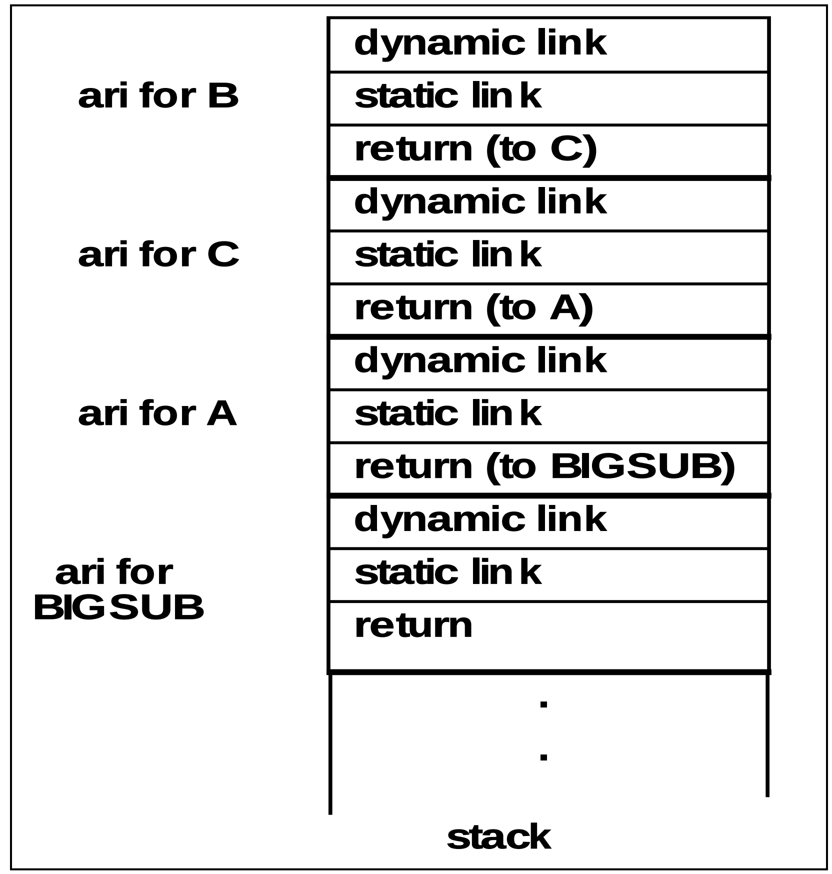
Stack contents for a dynamic-scoped program



ARI = activation record instance

Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position 1 in the following skeletal program.

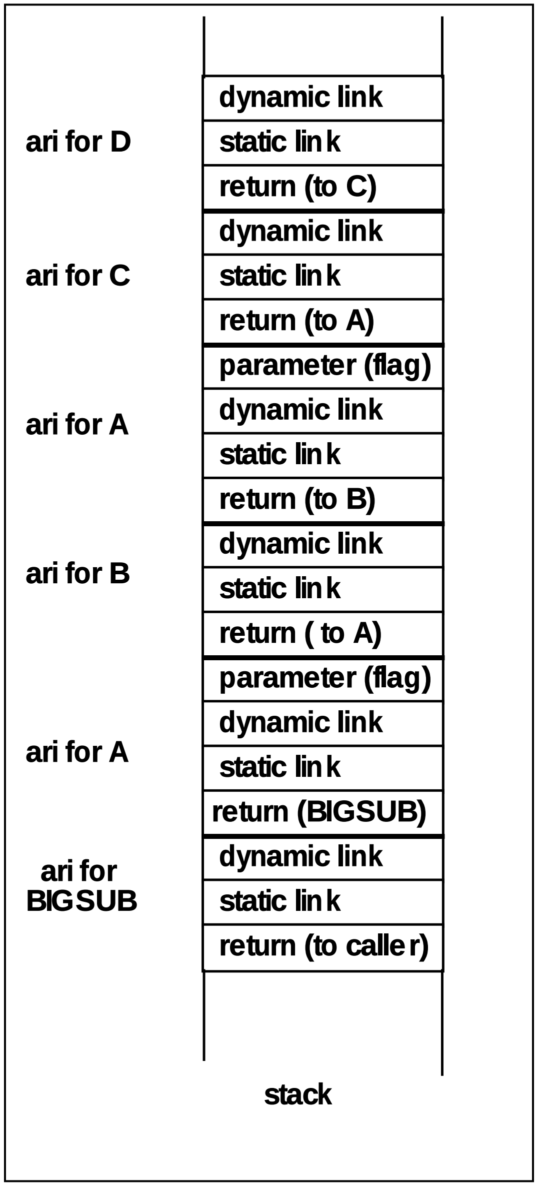
```
program MAIN;
      var X : integer;
   procedure Bigsub is
      procedure A is
         procedure B is
            begin    --- of B
            ……    ←———————— 1
            end;   ---- of B
         procedure C is
            begin  --- of C
            ……
            B
            end;  --- of C
         begin  --- of A
            ……
            C;
         end;  --- of A
  begin  ---- of Bigsub
      ……
        A;
  end --- of Bigsub
begin
  BIGSUB;
end; { MAIN }
```



ari: activation record instances

Show the stack with all activation record instances when execution reaches position 1

```
program MAIN;
          var X : integer;
    procedure Bigsub is
        procedure A(flag : Boolean) is
             procedure B is
                  begin     --- of B
                    ……
                   A(false)
                  end;    ---- of B
             begin   --- of A
                   if flag
                     then B;
                   else C;
            end;   --- of A
          procedure C is
             procedual D is
                  ……   <———————— 1
                end;   --- of D
             ……
             D;
           end;  ----- of C
    begin   ---- of Bigsub
         A(true) ;
    end --- of Bigsub
begin
   BIGSUB;
 end; { MAIN }
```



Chapter 11 Abstract Data Types and Encapsulation Constructs

- Abstract Data Type
    - Iterators of Collection ADT
- Parameterized Abstract Data Types
- Encapsulation Constructs
- Naming Encapsulations

Abstract Data Types

Abstract data type (ADT) is a set of data and the set of operations that can be performed on the data.

Built-in ADTs

boolean

– Values: true and false

– Operations: **and, or, not,** etc.

integer

– Values: Whole numbers between MIN and MAX values

– Operations: **add, subtract, multiply, divide**, etc.

arrays

– Values: Homogeneous elements, i.e., array of X. . .

– Operations: **initialize, store, retrieve, copy**, etc.

User-defined ADTs invokes operations on the data

Allows us to extend the programming language with new data types.

stack, symbol table, account, polynomial, matrix…

- The choice of what ADT to create depends on the application

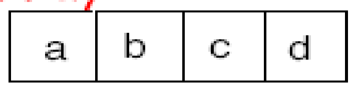
Compiler writing: tables, stacks, …
Banking: accounts, customers, …
Mathematical computing: matrices, sets, polynomials, …

- The choice of operations of the ADT depends on how you want to manipulate the data

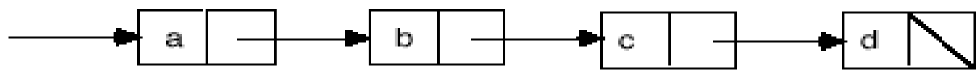
  Bank accounts: open, close, make a deposit, make a withdrawal, check the balance, …

- Repesentation of data: the data as represented in the computer

- Example: A "list" information structure, to give a few of many possible data structures:
  - could be an array



  - or could be a linked list

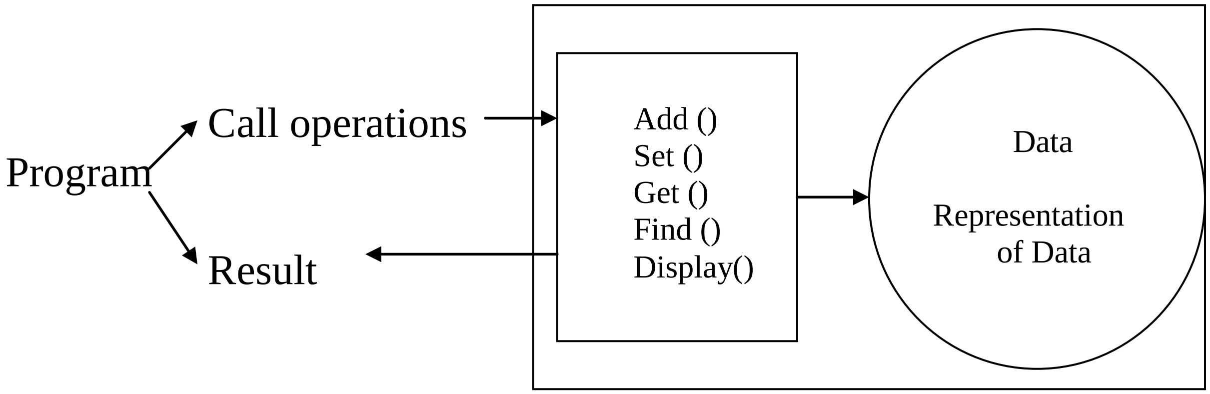


- Each of these is an *representation* or *implementation* of the abstraction.

• Array is better for:
–Accessing a randomly desired element

• Linked list is better at:
–Inserting
–Deleting
–Dynamic resizing

- Using ADT, we don’t need to care about the representation of objects (linked list, array, ets), we want to hide all the details about how dates are represented and access the object through the methods.



The advantages of ADT

- extend the programming language with new data type.
- valuable during problem modification and maintenance.

ADT for Stack and Queue

Stack

push: add info to the data structure
pop: remove the info MOST recently added
initialize, test if empty

Queue

put: add info to the data structure
get: remove the info LEAST recently added
initialize, test if empty

Could use EITHER array or "linked list" to implement EITHER stack or queue.

Iterator

A generalization of the iteration mechanism available in most programming languages.

- Provide a way to access each item in a collection ADT (Arraylist, List, Tree, etc)

The code of iterator contains a looping structure like follows,

for each *item i produced by iterator A*
do *perform some action on i*

To iterate through List L:

```
import java.util.*
Iterator it = L.iterator(); //create the Iterator
while (it.hasNext()) {     // see if finished
    Object ob = it.next(); // get the next item
    …
}
```

Iterators support abstraction by hiding how elements are produced.

- The user don't need to know the data type, and the collection type, vector, array or list, the user only choose a suitable iterator to access every element of the collection.
- Iterator is defined as an interface in Java, it returns a generator object.
- The generator's type is a subtype of Iterator. Different kinds of generator may use the same Iterator interface but different generators and different hasNext() and next() methods.

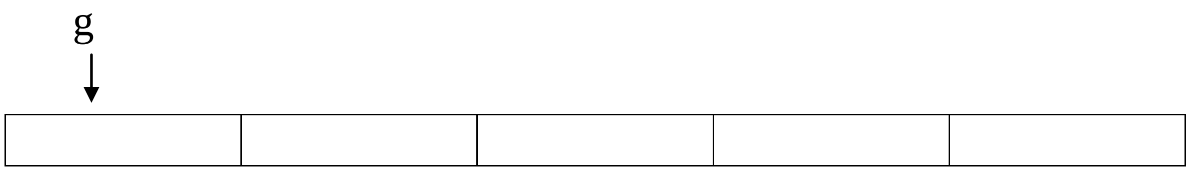
```
public interface Iterator {
  public boolean hasNext ( );
  // Returns true if there are no more elements else returns false

  public Object next ( ) throws NoSuchElementException;
   // If there are more results to yield, returns. The next result and modifies the state of
this to record the yield. Otherwise, throws NoSuchElementException
```

Specifying Iterators

```
public class IntSet {
  public Iterator elements ( )
   // Returns a generator that will produce all elements of this (as Integers), each exactly once,
   // in arbitrary order.
```

Using Iterators

```
public static int setSum (IntSet s) {
Iterator g = s.elements ( );
        int sum = 0;
        while (g.hasNext( ))
                sum = sum + ((Integer) g.next( )) . intValue;
        return sum;}
```



```
public static int max (Iterator g) throws EmptyException, NullPointerException {
//if g is null throw NullPointerException; if g is empty, throws EmptyException; else consumes all
element of g and returns the largest int in g.
try {
  int m= ((Integer)g.next()).intValue();
  while (g.hasNext()) {
    int x= g.next();
      if (m<x) m=x;;}
  return m; }
Catch (NoSuchElementException e)
   { throw new EmptyException (“Comp.max); }}}
```

Inner class in Java

(1) In Java, an inner class is a class nested within another class:

```
class C {

    class D {
    }
}
```

(2) Objects of the inner class are attached to objects of the outer class

You can't have an instance of the inner class without an instance to the outer one. This reference will keep the outer class instance around as long as the inner class instance exists. An instance of an inner class can only live attached to an instance of the outer class:

```
C c = new C()
D d = c.new D()
```

(3) The inner class is considered part of the implementation of the outer class, it has access to *all* of the outer class's instance variables and methods.

Implementing Iterators

To implement an iterator, one needs to write its code *and* define a class for its generator.

- An Iterator's implementation requires a class for the associated generator
- The generator class is a static inner class: it is nested inside the class containing the iterator and can access the private information of its containing class
- The generator class defines a subtype of the Iterator interface

```
Public class IntSet {
   private Vector els;
   public Iterator elements ( ) { return new IntGenerator (this); }
   // inner class
   private static class IntGenerator implements Iterator {
       private IntSet s;  // the IntSet being iterated
       private int n;  // index of the next element to consider

       IntGenerator (IntSet is) {
             s = is;
             n = 0;
       }
```

```
public boolean hasNext ( ) { return n < els.size(); }
public Object next ( ) throws NoSuchElementException {
            if ( n < s.els.size() ) {
               Integer result = s.els.get( n );
               n++;
               return result;
            } else
               throw NoSuchElementException("IntSet.elements");}
}       // s can access the private variable els from its outer class.
```