Pointers and RecursiveTypes

- A recursive type is one whose objects may contain one or more references to other objects of the type
- Records, List. Trees are some examples
- **In reference model,**
- a record of type foo to include a reference to another record of type foo
- **In value model,**
- recursive types require a pointer
- A variable whose value is a reference to some object

Syntax and Operations

- Operations on pointers include allocation and deallocation of objects in the heap,
- Dereferencing of pointers to access the objects
- Assignment of one pointer into another.

- The behavior of these operations depends heavily on
  - Language is functional or imperative
  - Reference or value model for variables/names.

- Functional model use a reference model for names
- Variables in imperative language use either value or reference model

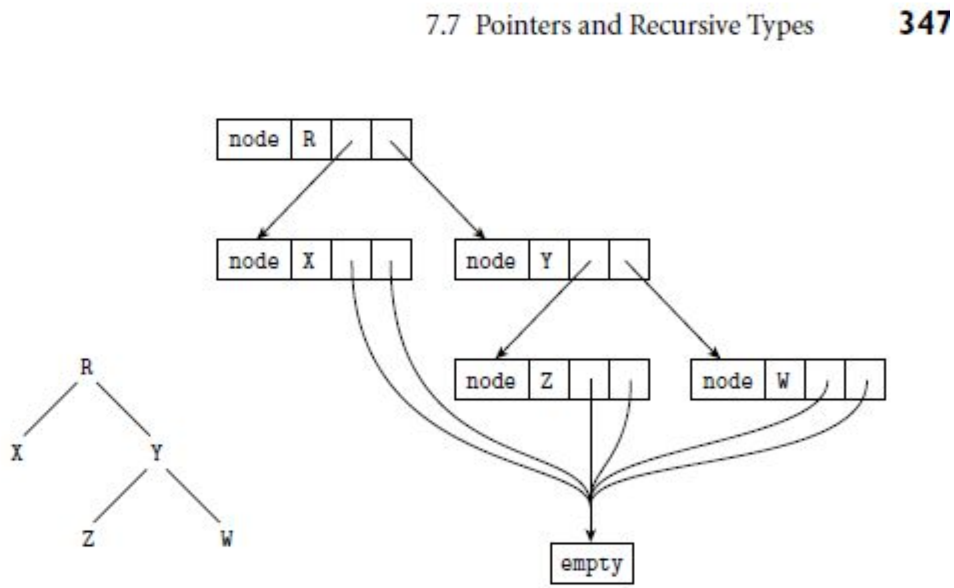
  A:=B
- C,Pascal,Ada
- puts value of B to A
- Clu and Smalltalka refers to same object to which B refers

- Java takes intermediate form
  - Built in types use value model
  - User defined types use reference model
- C# mirrors java by default

Reference Model

- In ML, the datatype mechanism can be used to declare recursive types
- datatype chr_tree = empty | node of char * chr_tree * chr_tree;
- chr_tree is either an empty leaf or a node consisting of a character and two child trees.

Tree implementation in ML



(#"R", node (#"X", empty, empty),
node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))

Lisp

- Tree is specified textually as

(#\R (#\X ()()) (#\Y (#\Z ()()) (#\W ()())))

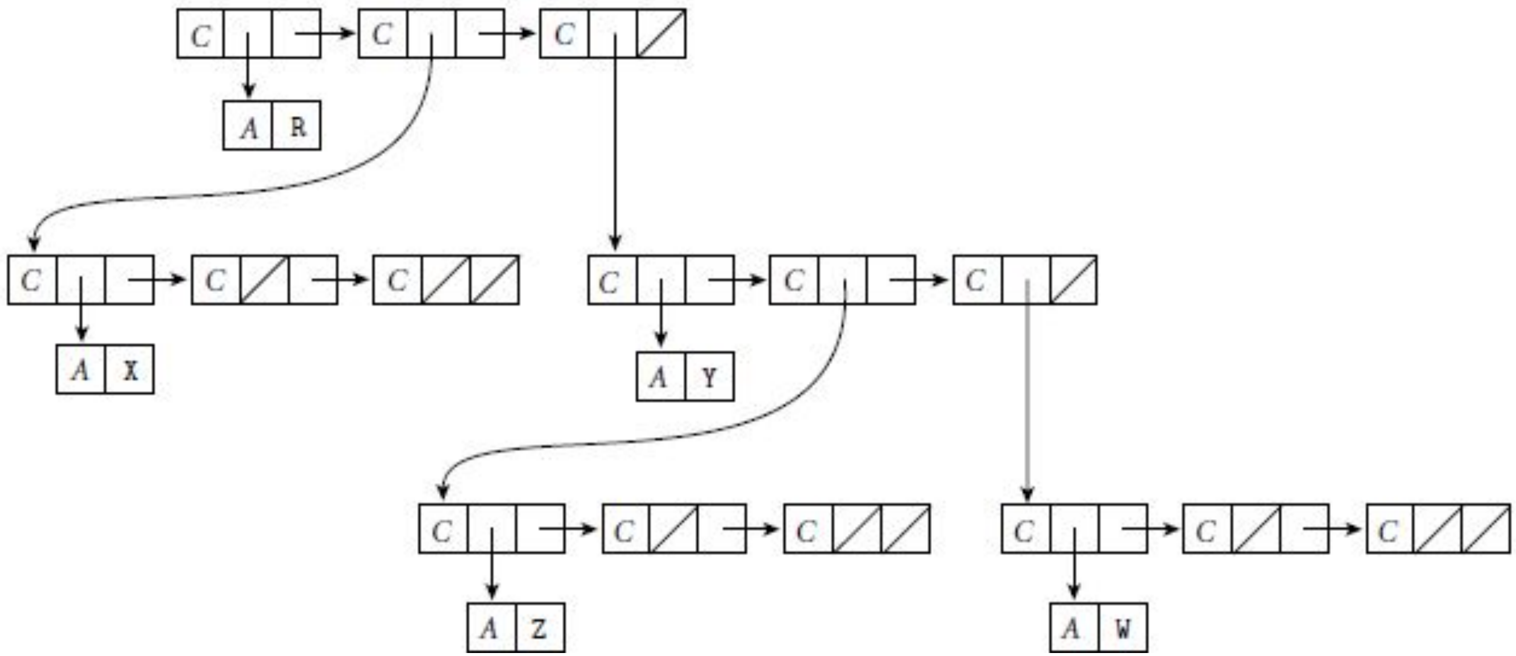
Outer most list contains three elements

Character R

Nested lists to represent left and right subtrees

- List are built using con cells
- At the top 1st con cell points to R
- 2nd and 3rd points to nested list representing left and right subtrees
- Each block of memory is tagged to indicate whether it is a cons cell or an atom.

Tree implementation in Lisp



Value Model

- In Pascal

```
type chr_tree_ptr = ˆchr_tree;
chr_tree = record
left, right : chr_tree_ptr;
val : char
end;
```

Value Model

- In Ada

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
left, right : chr_tree_ptr;
val : character;
end record;
```

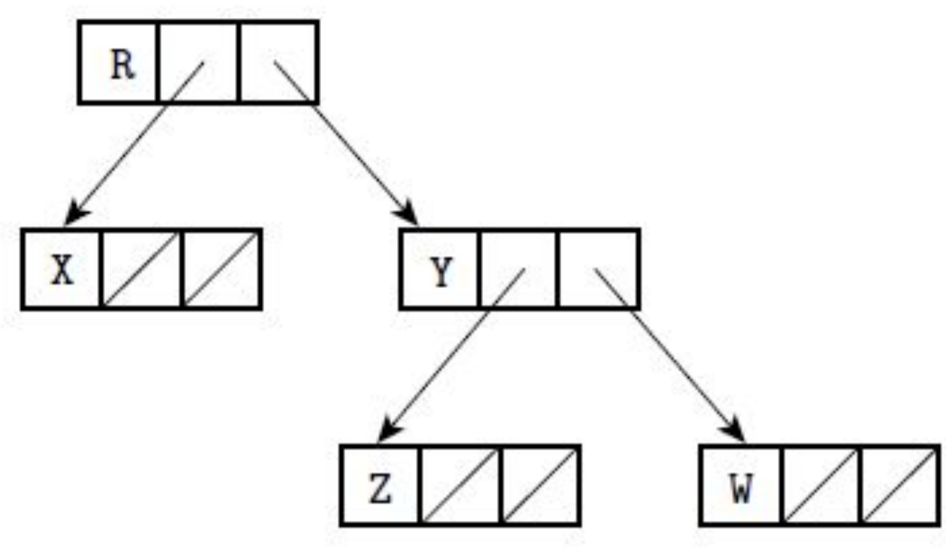
Value Model

- In C,

```
struct chr_tree {
struct chr_tree *left, *right;
char val;
};
```

- A tree must be created node by node
- To allocate a new node
- In Pascal
- new(my_ptr);
- In Ada
- my_pt:=new chr_tree;
- In C
- my_ptr=malloc(sizeof(struct chr_tree);

Pointer based tree



Pointer Dereferencing

- In Pascal and Modula: Postfix up arrow is used
- my_ptr^.val := 'X';
- In C. prefix * is used
- (*my_ptr).val = 'X';
- C also provides a postfix “right-arrow”
- my_ptr->val = 'X';

Pointer Dereferencing

- Ada dispenses dereferencing
- Same dot-based syntax can be used to access either a field of the record foo or a field of the record pointed to by foo, depending on the type of foo:

```
T : chr_tree;
P : chr_tree_ptr;
...
T.val := 'X';
P.val := 'Y';
```

Pointers and Arrays in C

- Pointers and arrays are closely linked in C.

```
int n;
int *a;
int b[10];
```

Array name is automatically converted to a pointer to arrays 1st element

- 1. a = b;
- 2. n = a[3];
- 3. n = *(a+3); /* equivalent to previous line
- 4. n = b[3];
- 5. n = *(b+3) /* equivalent to previous line

Multidimentional arrays

- int *a[n]
- Allocates space for n row pointers
- allocate space for a 2D array
- int a[n][m]

Multidimentional arrays

- when an array is used as parameter in function call
  - For 1 D array formal parameter is declared as
  - int a[] or int*a
  - For 2 D array, formal parameter with row pointer layout is declared as
  - int *a[ ] or int**a
  - For 2 D array, formal parameter with comtiguusos layout is declaredas
  - int a [][m] or int(*a)[m]

Dangling References

- When a heap-allocated object is no longer live, object's space must be reclaimed
- Stack objects are reclaimed automatically by epilogue
- Heap objects are reclaimed in 2 ways

- In Pascal,C, and C++ require the programmer to reclaim an object explicitly

Dangling References

- In Pascal:

  dispose(my_ptr);
- In C:

  free(my_ptr);
- In C++:

  delete my_ptr;

Using automatic garbage collector

Example

- A *dangling reference* is a live pointer that no longer points to a valid object

```
int i = 3;
int *p = &i;
...
void foo() { int n = 5; p = &n; }
...
cout << *p; // prints 3
foo();
...
cout << *p; // undefined behavior: n is no longer live
```

```
int *p = new int;
*p = 3;
...
cout << *p; // prints 3
delete p;
...
cout << *p; // undefined behavior: *p has been reclaimed
```

Garbage Collection

- Explicit reclamation of heap objects may create bugs
- memory leaks
- dangling references

- Garbage Collection
- Language implementation notice when objects are no longer useful and reclaim them automatically

Reference Counts

- When is an object no longer useful?
  - when no pointers to it exist
  - simplest garbage collection technique
- places a counter in each object
- Counter track of the number of pointers that refer to the object

- When the object is created, this reference count is set to 1
- to represent the pointer returned by the new operation

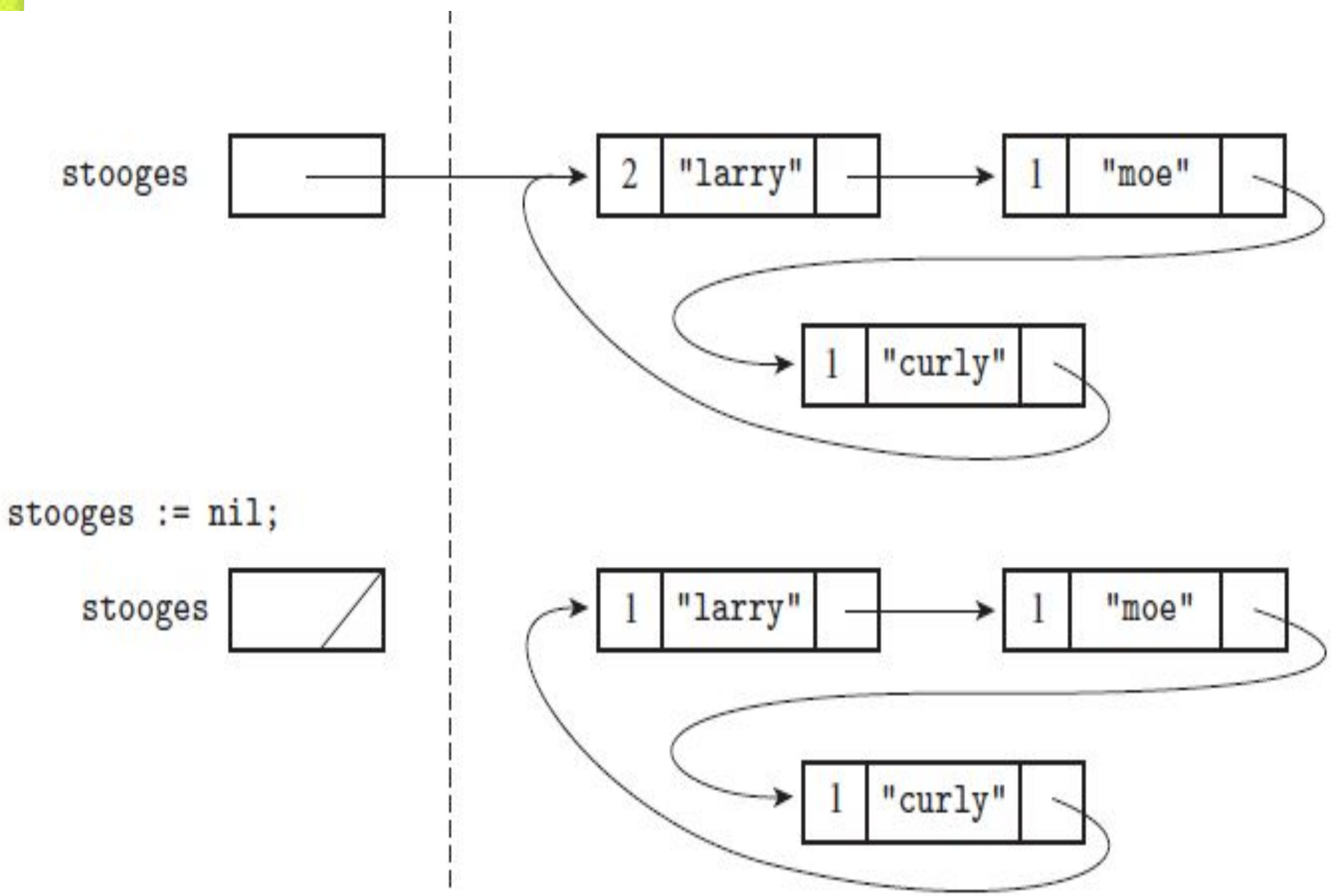
- When a pointer is assigned to another
- String S1=new String()
- Sets refcount of S to 1

- When a pointer is asigned into anotther
- String S1=new String() refcount of S1= 1
- String S2=S1 refcount of S1= 2

- refcount of S2 id decremented by 1 and
- refcount of S1 is incremented by 1

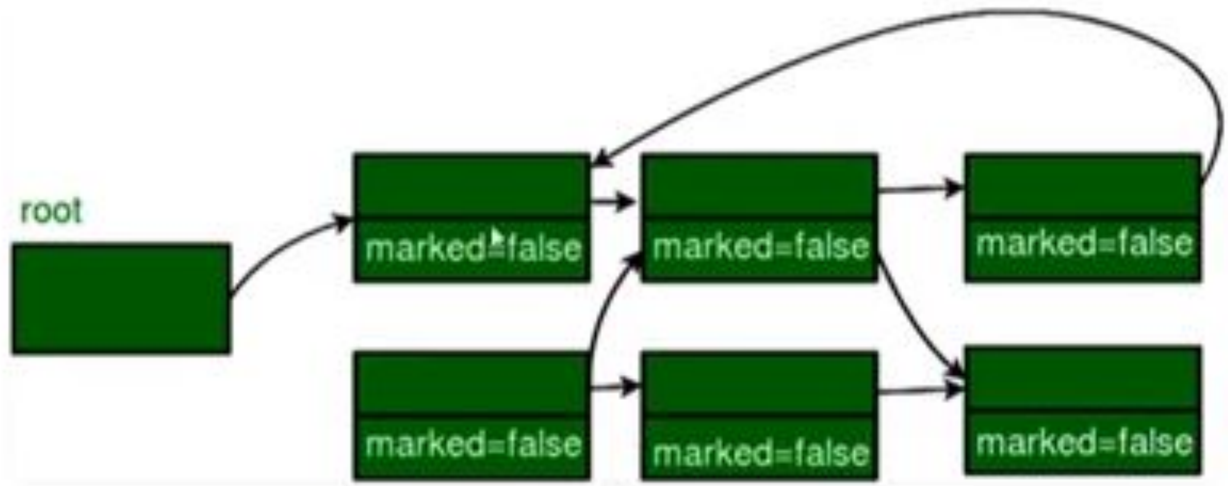
- When a reference count reaches zero, its object can
- be reclaimed
- The run-time system must decrement counts for any
- objects referred to by pointers within the object being reclaimed
- If a,b,c,d are pointers with refcount=1
- respectively
- a=b refcount of a=0 and b=2
- c=a refcount of c=0

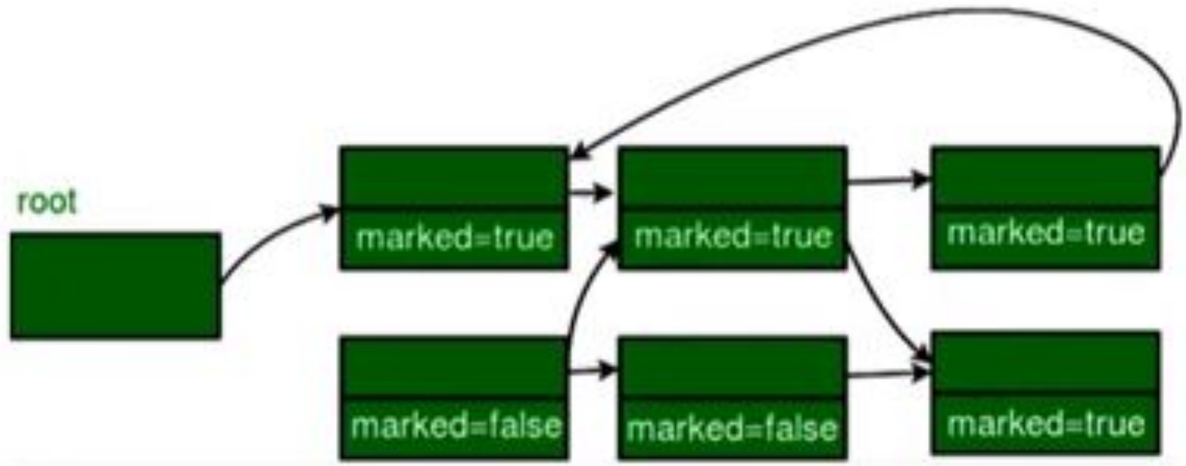
- c refers to a whose refcount is decremented
- Reclaim those objects whose refcount =0
- Reclaim the space of a and c
- This is a cascading effect



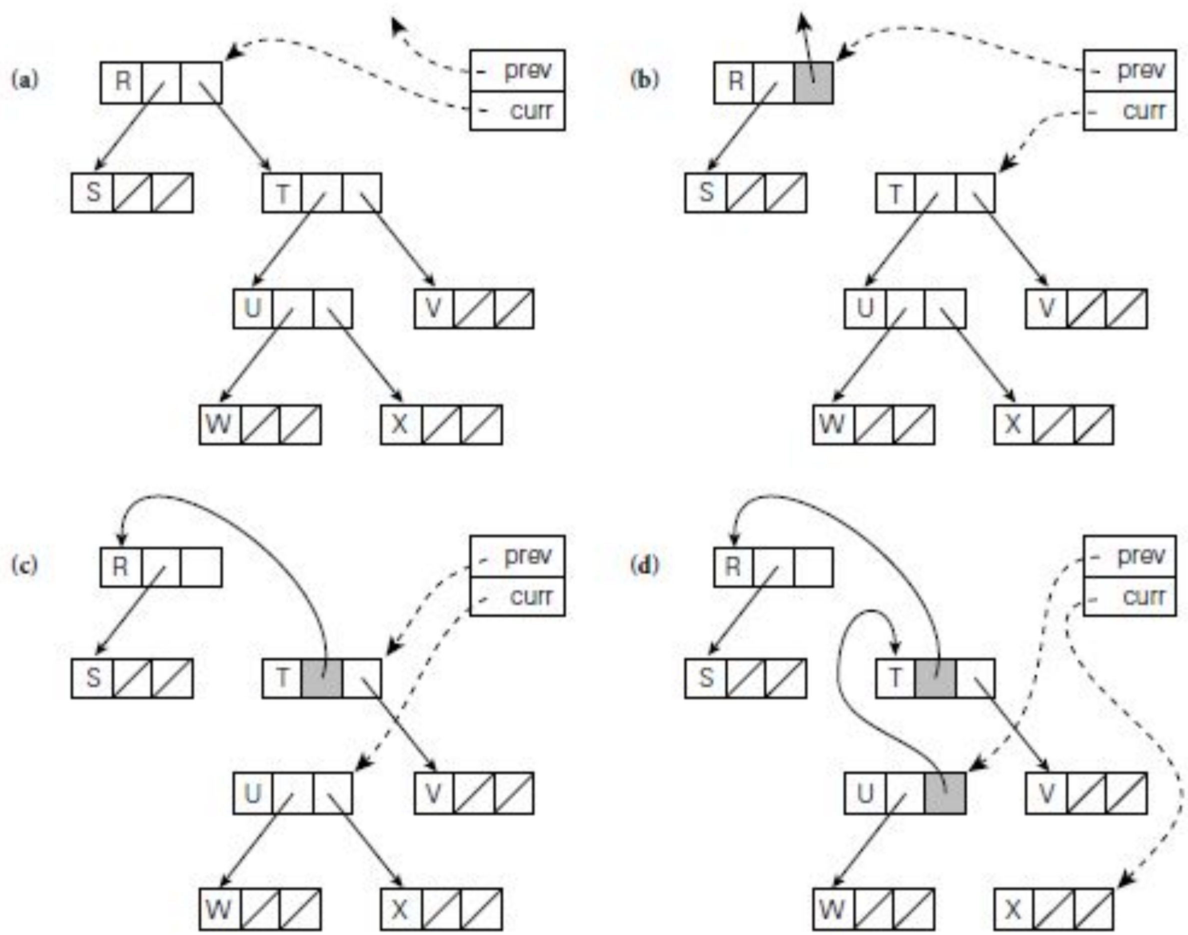
stooges
2
"larry"
1
"moe"
1
"curly"
stooges := nil;
stooges
1
"larry"
1
"moe"
1
"curly"

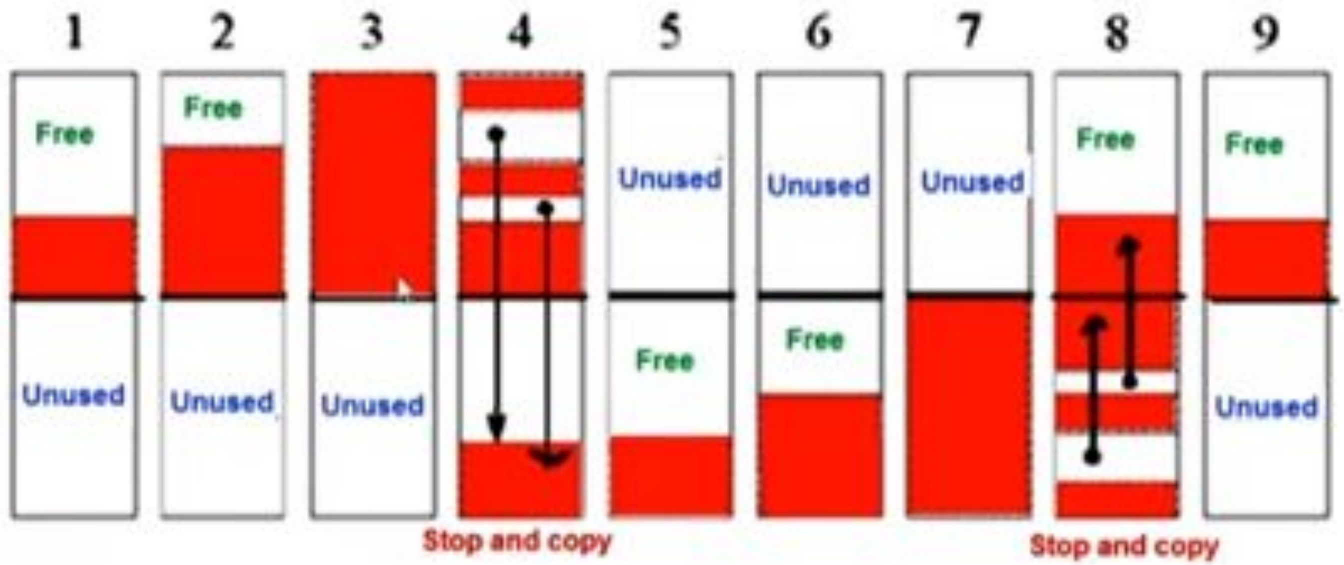
Mark and sweep




root
marked=true
marked=true
marked=true
marked=false
marked=false
marked=true

Pointer Reversal




1
2
3
4
5
6
7
8
9
Free
Free
Free
Free
Unused
Unused
Unused
Unused
Unused
Unused
Unused
Free
Free
Stop and copy
Stop and copy

LIST

LIST

- A list is an abstract data type that represents a countable number of ordered values, where same value may occur more than once.

- Like arrays, contain a sequence of elements, but there is no notion of mapping or indexing

- A list is defined recursively as either an empty list or a pair consisting of a head element and a reference to a sublist
- Items in the list can be either empty or atom(An obj of built in type)
- ML,Lisp,Python are the languages using List

- Homogeneous List --> All elements of same type(ML)
- Heterogonous List --> All elements of different type (Lisp List)

- List in [ ] square brackets - ML, comma seperated elements ( ) parathesis - Lisp, separated by white spaces

- An ML list is usually a chain of blocks, each of which contains an element and a pointer to the next block
- A Lisp list is a chain of cons cells, each of which contains two pointers, one to the element and one to the next cons cell

- ML- [a, b, c, d].
- Lisp - (a b c d)
- A list may contain duplicate values

- In Lisp,
- cons- constructs a list
- car and cdr – to extract a part of list

In Lisp:

(cons 'a '(b)) =⇒ (a b)

(car '(a b)) =⇒ a

(car nil) =⇒ ??

(cdr '(a b c)) =⇒ (b c)

(cdr '(a)) =⇒ nil

(cdr nil) =⇒ ??

(append '(a b) '(c d)) =⇒ (a b c d)

In ML the equivalent operations are written as follows:

Basic list operations in ML

a :: [b] =⇒ [a, b]

hd [a, b] =⇒ a

hd [ ] =⇒ run-time exception

tl [a, b, c] =⇒ [b, c]

tl [a] =⇒ nil

tl [ ] =⇒ run-time exception

[a, b] @ [c, d] =⇒ [a, b, c, d]

SETS

- A programming language set is an unordered collection of an arbitrary number of distinct values of a common type.
- Sets were introduced by Pascal, and are found in many more recent languages as well

- The type from which elements of a set are drawn is known as the *base* or *universe* type.

- Pascal supports sets of any discrete

type, and provides union, intersection, and difference operations.

- var A, B, C : set of char;
- D, E : set of weekday;
- ...
- A := B + C; (* union; A := {x | x is in B or x is in C} *)
- A := B * C; (* intersection; A := {x | x is in B and x is in C} *)
- A := B - C; (* difference; A := {x | x is in B and x is not in C} *)

Strings

- A string is simply an array of characters
- powerful string facilities are found in Snobol, Icon, and the various scripting languages.
- Literal strings to be specified as a sequence of characters, usually enclosed in single or double quote marks

- Most languages also provide *escape sequences* that allow nonprinting characters and quote marks to appear inside of strings

- Several languages that do not in general
- allow arrays to change size dynamically do provide this flexibility for strings

- First, manipulation of variable-length strings is fundamental to a huge number of computer applications, and in some sense “deserves” special treatment.

- Second, the fact that strings are one-dimensional, have one-byte elements,

  and never contain references to anything else makes dynamic-size strings easier to implement than general dynamic arrays.

FILES

- Input/output (I/O) facilities allow a program to communicate with the outside world
- Files generally refer to off-line storage implemented by the operating system.
- Files may be further categorized into those that are **temporary** and those that are **persistent**

- Temporary files exist for the duration of a single program run

- Their purpose is to store information that is too large to fit in the memory available to the program.

Persistent files allow a program to read data that existed before the program began running, and to write data that will continue to exist after the program has ended

Equality Testing and Assignment

- Assignment operator (=) is a binary operator which operates on two operands that assigns the vlaue of right side expressions or variables value to the left side variable

- eg:
- x=(a+b)
- y=x;

- Equality (==) is a comparison operator. It is aslso a binary operator that operates on two operands, compares the value of left and right hand side expressions return 1 if they are equal else 0

- For simple, primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively straightforward

- But for abstract types,
- Equality/comparison can be
  - Shallow comparison
  - Deep comparison

- Under a reference model of variables, a shallow assignment a := b
- will make **a** refer to the object to which **b** refers.
- A deep assignment will create a copy of the object to which b refers, and make a refer to the copy.

- Under a value model of variables, a shallow assignment will copy the value of b into a,