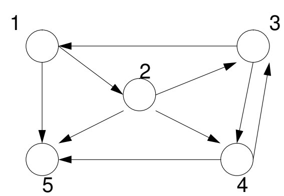
CSE 326
Lecture 3

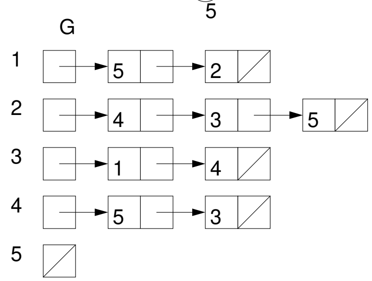
Spring Quarter 1998

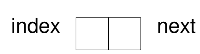
Multiple Linked Lists

- Many ADTs such as graphs, relations, sparse matrices, radix sort, multivariate polynomials use multiple linked lists.
- There are several options
  - array of lists
  - list of lists
  - cross lists

Adjacency List Representation of Directed Graph





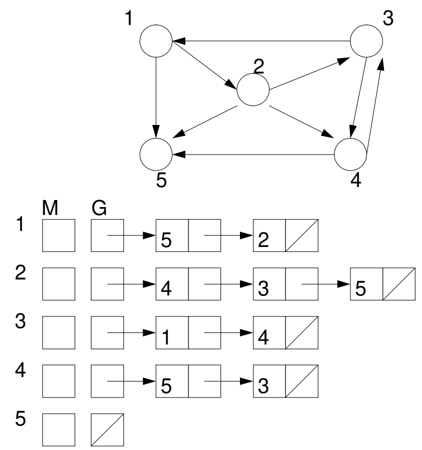


- Array G of unordered lists
- Each list entry corresponds to an edge in the graph.

Reachability by Marking

- Suppose we want to mark all the node in the graph which are reachable from a given node k. Let G[1..n] be the adjacency list representation of the graph and let M[1..n] be the mark array which is initialized to all 0.
- mark(i: integer) {

```
 x: local node pointer;
M[i] := 1;
x := G[i];
while not(x=null) do {
   if M[x.index] = 0 then mark(x.index);
   x := x.next}}
```



mark(1)

Notes on Reachability

- The marking algorithm visits each node and each edge at most once.
- This marking algorithm is commonly called depth-first-search (DFS) and variants are used in solving many problems.
- Graph reachability is closely related to garbage collection.

  - the nodes are blocks of memory.

  - the marked blocks are those which are reachable from a variable in the program. The unmarked blocks are garbage.

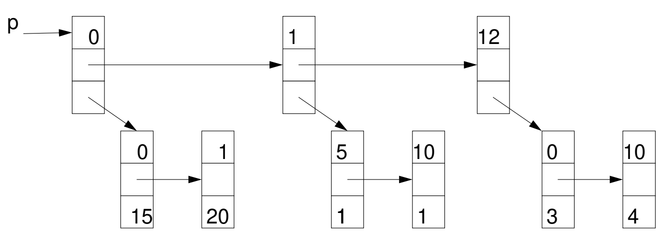
  - In real application where the number of successors is bounded the recursion and any stack can be eliminated. This is called the Deutch-Schorr-Waite algorithm, circa 1965.

  - Garbage collection can be done using reachability (depth-first search).

Multivariate Polynomial as a List of Lists

- $15 + 20y + xy^{10} + xy^5 + 3x^{12} + 4x^{12}y^{10} =$

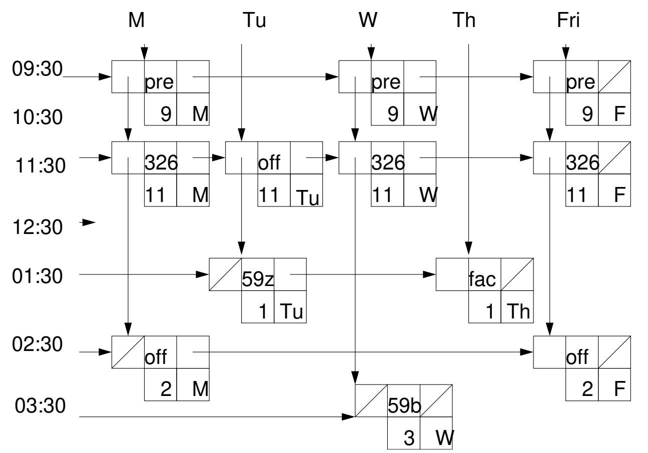
  $(15 + 20y)x^0 + (y^5 + y^{10})x^1 + (3 + 4y^{10})x^{12}$



Daily Schedule Cross List

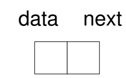
	M	Tu	W	Th	Fri
09:30	prep		prep		prep
10:30					
11:30	326	off	326		326
12:30					
01:30		590z		fac	
02:30	off				off
03:30			590b		

Cross List



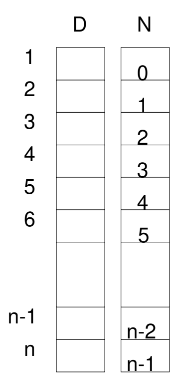
Implementing Records and Pointers using Arrays

- This will be needed in many languages such as Fortran and Basic.
- It is very simple when the number records that will be needed is known in advance.
- Consider a very basic node type with two fields "data" (integer) and "next" (pointer).



- We know that no more than n records will ever be needed at one time.
- We declare two arrays D[1..n] and N[1..n] each which holds an integer. D is initialize to 0 and N is initialized such that N[k] = k-1 for k = 1 to n.
- We also declare an integer variable Free which will always be in the range 0 to n. Free is initialized to n.

Initial Data Structure

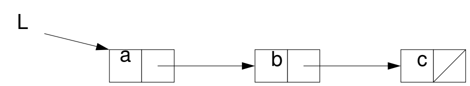


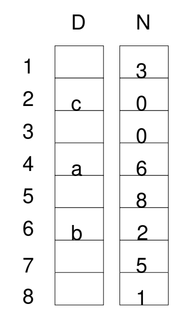
Free = n

Basic Correspondences

- pointer = some integer from 0 to n
- null = 0
- p.data = D[p]
- p.next = N[p]
- The index Free is the pointer to the "list of free nodes".

Data Structure in Use





L = 4

Free = 7

- Insert(x) {

  local integer q;
  if not(Free = 0) then q := Free
  else "overflow";
  Free := N[Free];
  D[q] := x;
  N[q] := L;
  L := q }

Data Structure in Use (Cont.)

	D	N
1		3
2	c	0
3		0
4	a	6
5		8
6	b	2
7		5
8		1

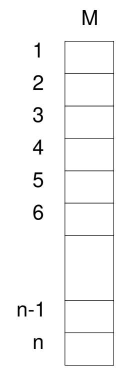
L = 4

Free = 7

- Delete_First {

  local integer q;
  q := L;
  L := N[L];
  N[q] := Free;
  Free := q }

Global Memory Management



- M is the virtual address space.
- An object that requires m memory locations will take the block M[k], M[k+1], ... , M[k+m-1].
- Extra storage might be necessary for marking the block for memory management functions.
- Typical memory management functions

  -keeping a "heap" of free unstructured storage.

  - keeping free lists of objects of the same type.

  - moving unused objects back to the heap, perhaps merging objects of the same or different type.

  - garbage collection.