Data Structures: Lists

Wei-Mei Chen

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

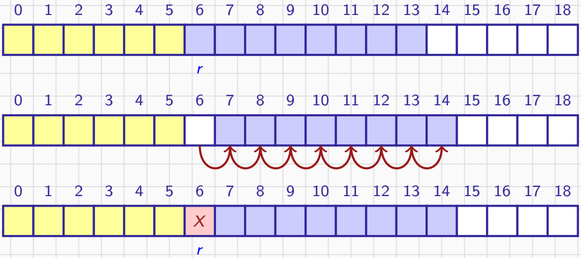
Linked Lists

Introduction

Array: successive items locate a fixed distance

☞ Disadvantages:

- data movements during insertion and deletion
- waste space in storing n ordered lists of varying size



Nonsequential List-Representation

Insert "GAT" at Position 5

first → 8

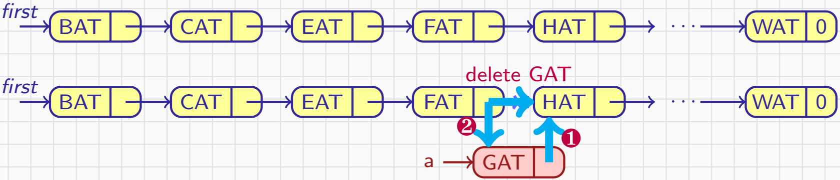
	1	2	3	4	5	6	7	8	9	10	11
	HAT		CAT	EAT			WAT	BAT	FAT		VAT
	11		4	9			0	3	1		7

first → 8

	1	2	3	4	5	6	7	8	9	10	11
	HAT		CAT	EAT	GAT		WAT	BAT	FAT		VAT
	11		4	9	1		0	3	5		7

A Linked List

1. Get a node a
2. Set the "data" field of a to "GAT"
3. Set the "link" field of a to point to the node after FAT, which contains HAT
4. Set the "link" field of the node containing FAT to a



Representations

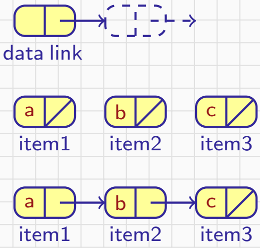
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
char data;
struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



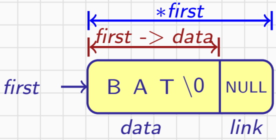
List of Words

- A node:

```
typedef struct listNode *listPointer;
typedef struct listNode {
    char data[4];
    listPointer link;
};
```

- Creation

```
listPointer first = NULL;
```



- Testing

```
#define IS_EMPTY(first) (!(first))
```

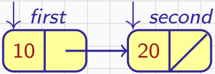
- Structure member: `(*e).name` OR `e->name`
- Copy

```
strcpy(first->data,"BAT");
first->link = NULL;
```

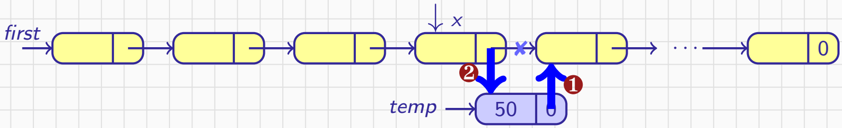
Two-Node Linked List

Create a two-node list

```
typedef struct listNode *listPointer;
typedef struct listNode {
    int data;
    listPointer link;
} listNode;
listPointer first =NULL;
listPointer create2( ) {
    listPointer first, second;
    MALLOC(first,sizeof(listNode));
    MALLOC(second,sizeof(listNode));
    second->link = NULL;
    second->data = 20;
    first->data = 10;
    first->link = second;
    return first;
}
```



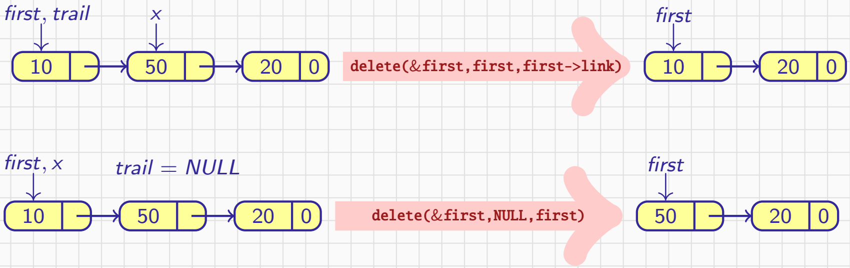
Singly Linked Lists: Insertion




```
void insert(listPointer *first, listPointer x)
{  /*Insert a node after node x
    listPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = 50;
    if (*first) {
        temp->link = x->link;
        x->link = temp;
    }
    else {
        temp->link = NULL;
        *first = temp;
    }
}
```

```
insert(&first, x);
```

Singly Linked Lists: Deletion



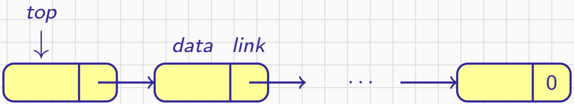

```
void delete(listPointer *first, listPointer trail, listPointer x)
{  /*Delete node x
   if (trail) /*Delete a node other than the first node
      trail->link = x->link;
    else /*Delete the first node
      *first = (*first)->link;
    free(x);
}
```

Printing out a List

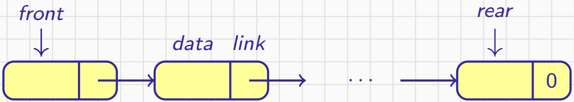
```
void printList(listPointer first)
{
   printf("The list contains:  ")
   for (; first; first = first->link)
      printf("%4d", first->data);
   printf("\n");
}
```

Linked Stacks and Queues

❇**Linked stack**:



❇**Linked queue**:



Multiple Stacks

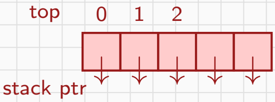
```
#define MAX_STACKS 10
typedef struct element {
    int key;
    /* other fields */
} element;
typedef struct stack *stackPointer;
typedef struct stack {
    element data;
    stackPointer link;
} stack;
stackPointer top[MAX_STACKS];
```

Initial condition:

top[i] = NULL, $0 \le i <$ MAX

Boundary condition:

top[i] = NULL $\Leftrightarrow$ ith stack is empty




```
void push(int i, element item)
{   stackPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = item;
    temp->link = top[i];
    top[i] = temp;
}
```

```
element pop(int i)
{   stackPointer temp = top[i];
    element item;
    if (!temp)
        return stackEmpty();
    item = temp->data;
    top[i] = temp->link;
    free(temp);
    return item;
}
```

Multiple Queues

```c
#define MAX_QUEUES 10
typedef struct queue
*queuePointer;
typedef struct queue {
    element data;
    queuePointer link;
} queue;
queuePointer front[MAX_QUEUES],
             rear[MAX_QUEUES];
```

Initial condition:

front[i] = NULL, $0 \le i <$ MAX

Boundary condition:

front[i] = NULL $\Leftrightarrow$ ith queue is empty

```c
void addq(int i, element item)
{   queuePointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = item;
    temp->link = NULL;
    if (front[i])
        rear[i] ->link = temp;
    else
        front[i] = temp;
    rear[i] = temp;
}
```

```c
element deleteq(int i)
{   queuePointer temp = front[i];
    element item;
    if (!temp)
        return queueEmpty();
    item = temp->data
    front[i] = temp->link;
    free(temp);
    return item;
}
```

Polynomials

Polynomial Representation

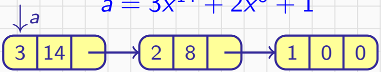
- $A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$
  where $a_i \neq 0$ for all i and $e_{m-1} > \cdots \geq 0$
- Assume that $a_i \in \mathbb{Z}^+$

```
typedef struct polyNode
*polyPointer;
typedef struct polyNode {
    int coef;
    int expon;
    polyPointer link;
} polyNode;
polyPointer a,b;
```

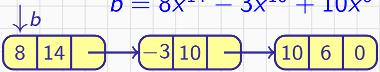
coef	expon	link

Example

$a = 3x^{14} + 2x^8 + 1$



$b = 8x^{14} - 3x^{10} + 10x^6$



```c
polyPointer padd(polyPointer a, polyPointer b)
{   polyPointer c, rear, temp;
    int sum;
    MALLOC(rear, sizeof(*rear));
    c = rear;
    while (a && b)
        switch (COMPARE(a->expon,b->expon)) {
            case -1:  /* a->expon < b->expon */
                attach(b->coef,b->expon,&rear);
                b = b->link;
                break;
            case 0:  /* a->expon = b->expon */
                sum = a->coef + b->coef;
                if (sum) attach(sum,a->expon,&rear);
                a = a->link; b = b->link; break;
            case 1:  /* a->expon > b->expon */
                attach(a->coef,a->expon,&rear);
                a = a->link;
        }
    for (; a; a = a->link) attach(a->coef,a->expon,&rear);
    for (; b; b = b->link) attach(b->coef,b->expon,&rear);
    rear->link = NULL;
    temp = c; c = c->link; free(temp);
    return c;
}
```

Attach a Node

```c
void attach(float coefficient, int exponent, polyPointer *ptr)
{
    polyPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->coef = coefficient;
    temp->expon = exponent;
    (*ptr)->link = temp;
    *ptr = temp;
}
```

Analysis of padd

- There are three cost measures
  1. coefficient additions
  2. exponent comparisons
  3. creation of new nodes for c
- Assume that each of these operations takes a single unit of time if done once. And
  $A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$ and
  $B(x) = b_{n-1}x^{f_{n-1}} + \cdots + b_0 x^{f_0}$
  $\Rightarrow 0 \leq \#\ of\ coefficient\ additions \leq \min\{m, n\}$
- We make one comp on each iteration of the `while` loop. On each iteration, either a or b or both move to the next term.
  An extreme case: $e_{m-1} > f_{n-1} > e_{m-2} > f_{n-2} > \cdots > e_0 > f_0$
  $\Rightarrow\ \#\ of\ exponent\ comparisons \leq m + n - 1$
- In summary, the max # of the executions $\Rightarrow O(m+n)$

Erasing Polynomials

- **Example:**

$$e(x) = a(x) * b(x) + d(x)$$

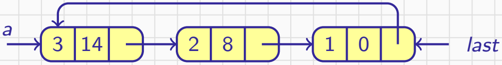
```
polyPointer a,b,d,temp;
a = readPoly();
b = readPoly();
d = readPoly();
temp = pmult(a,b);
e = padd(temp,d);
```

Return the nodes of temp(x)!!!

```
void erase(polyPointer *ptr)
{   polyPointer temp;
    while (*ptr) {
        temp = *ptr;
        *ptr = (*ptr) ->link;
        free(temp);
    }
}
```

- We can free all the nodes of a polynomial more efficiently if we modify our list structure to a circular list. (not a chain)

  EX: $3x^{14} + 2x^{8} + 1$



- We free nodes that are no longer in use. But we may reuse some nodes later.

  $\Rightarrow$ Collect the "freed" space to the *avail* list

  $\Rightarrow$ Only when the list is empty do we need to use "MALLOC" to

  create a new node.

Maintaining an Available List

Constant time:

Independent of # of nodes in a list $\Rightarrow O(1)$.

```
void retNode(polyPointer node)
{
    node->link = avail;
    avail = node;
}
```

Initially, `avail = NULL`

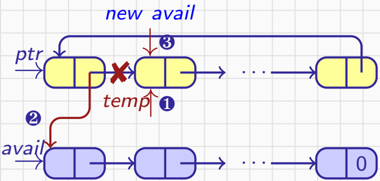
```
polyPointer getNode(void)
{   polyPointer node;
    if (avail) {
        node = avail;
        avail = avail ->link;
    }
    else
        MALLOC(node, sizeof(*node));
    return node;
}
```

Erasing a Circular List

- We may erase a circular list in a fixed amount of time

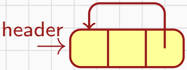
  constant time!

```
void cerase(polyPointer *ptr)
{   polyPointer temp;
    if (*ptr) {
        temp = (*ptr)->link;   ❶
        (*ptr)->link = avail;  ❷
        avail = temp;  ❸
        *ptr = NULL;
    }
}
```



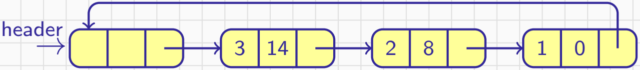
A Header Node

- Note: we have to handle **the zero polynomial** as a special case!
- A possible solution ➠ header nodes
  Each polynomial, zero or nonzero, contains a one additional node.



Zero polynomial

$3x^{14} + 2x^8 + 1$



```c
polyPointer cpadd(polyPointer a, polyPointer b)
{   polyPointer startA, c, lastC;
    int sum, done = FALSE;
    startA = a; a = a->link; b = b->link;
    c = getNode(); c->expon = -1; lastC = c;
    do {
        switch (COMPARE(a->expon, b->expon)) {
            case -1:  /* a->expon < b->expon */
                attach(b->coef,b->expon,&lastC);
                b = b->link;
                break;
            case 0:  /* a->expon = b->expon */
                if (startA == a) done = TRUE;
                else {
                    sum = a->coef + b->coef;
                    if (sum) attach(sum,a->expon,&lastC);
                    a = a->link; b = b->link;
                }
                break;
            case 1:  /* a->expon > b->expon */
                attach(a->coef,a->expon,&lastC);
                a = a->link;
        }
    } while (!done);
    lastC->link = c; return c;
}
```

Additional Operations

Inverting for Chains

- Chain: A singly linked list in which the last node has a null link

```
listPointer invert (listPointer lead)
{   listPointer middle, trail;
    middle = NULL;
    while (lead){
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```

$O(length)$

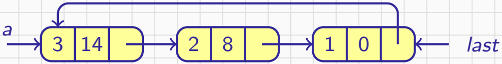
Concatenating Two Chains

- Concatenates two chains, ptr1 and ptr2.
- Assign the list ptr1 followed by the list ptr2.

```
listPointer concatenate (listPointer ptr1, listPointer
ptr2)
{   listPointer temp;
    if (!ptr1) return ptr2;
    if (!ptr2) return ptr1;
    for (temp = ptr1; temp->link; temp = temp->link) ;
    temp->link = ptr2;
}
```

Inserting at the Front of a Circularly List

- $3x^{14} + 2x^{8} + 1$




```
void insertFront (listPointer *last, listPointer node)
{
    if (!(*last)) {
        *last = node;
        node->link = node;
    }
    else {
        node->link = (*last)->link;
        (*last)->link = node;
    }
}
```

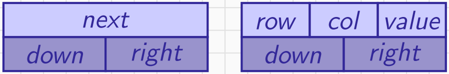
Finding the Length of a Circular List

```
int length (listPointer last)
{   listPointer temp;
    int count = 0;
    if (last) {
        temp = last;
        do {
            count++;
            temp = temp->link;
        } while (temp != last);
    }
    return count;
}
```

Sparse Matrices

Data Structures of Sparse Matrices

- Sequential representation of sparse matrix suffered from the same inadequacies as the similar representation of Polynomial.
- In Chapter 2, $< row, column, value >$
- Here we study a circular linked list representation of a sparse matrix.
  Two types of nodes:
  - head node: tag, down, right, and next
  - entry node: tag, down, row, col, right, and value

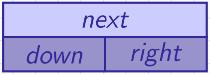


a head node   an entry node

Head Nodes for Sparse Matrices

Head node i is the head node for both row i and column i.

- Each head node involves three lists: a row list, a column list, and a head node list.
  - *down*: link into a column list
  - *right*: link into a row list
  - *next*: link into the head node together
- The total number of head nodes is $\max\{\#\ of\ rows, \#\ of\ cloumns\}$.



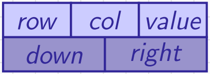
a head node

Entry Nodes for Sparse Matrices

If $a_{ij} \neq 0$, there is a node with

tag field $\leftarrow$ entry, value $\leftarrow a_{ij}$, row $\leftarrow i$, col $\leftarrow j$

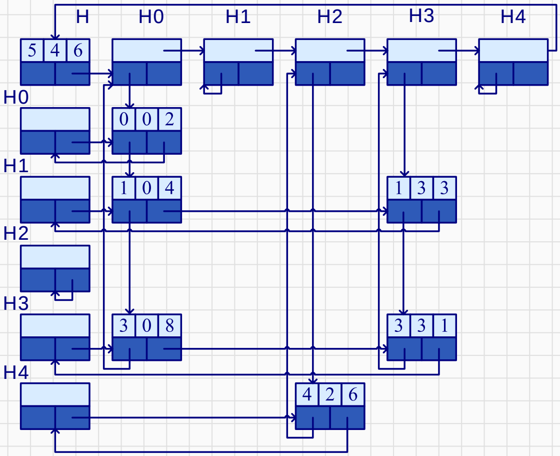
- Link this node into the circular linked list for row i and column j
  - *down*: link into the next nonzero term in the same col
  - *right*: link into the next nonzero term in the same row
- For an $n \times m$ sparse matrix with r nonzero terms, the number of nodes needed is $\max\{n, m\} + r + 1$.



an entry node

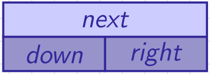
An Example

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

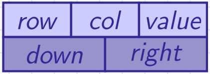


Structures of Sparse Matrices

```
#define MAX_SIZE 50
typedef enum {head, entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct entryNode {
    int row;
    int col;
    int value;
}
typedef struct matrixNode {
    matrixPointer down;
    matrixPointer right;
    tagfield tag;
    union {
        matrixPointer next;
        entryNode entry;
    } u;
};
matrixPointer hdnode[MAX_SIZE];
```



a head node



an entry node

Reading in a Matrix

- Assume: input $< row, col, value >$

	0	1	2
0	5	4	6
1	0	0	2
2	1	0	4
3	1	3	3
4	3	0	8
5	3	3	1
6	4	2	6

- -> *hdnode*

$hdnode[i]$ points to ith row and ith column

- Procedure `mread` reads in a sparse matrix.
  $O(\max\{numRows, numCols\} + numTerms) =$
  $O(numRows + numCols + numTerms)$

Erasing a Matrix

```
void merase(matrixPoiner *node)
{
    matrixPointer x, y, head=(*node)->right;
    int i;
    for(i=0; i < (*node)->u.entry.row; i++) {
        y = head->right;
        while(y!=head) {
            x =y; y = y->right; free(x);
        }
        x = head; head = head->u.next; free(x);
    };
    y= head;
    while(y != *node){
        x =y; y = y->u.next; free(x);
    }
    free(*node); *node = NULL;
}
```

$$O(numRows + numCols + numTerms)$$

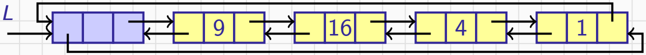
Doubly Linked Lists

- Singly linked list (in one direction only)
- Move in forward and backward direction.

  How to get the preceding node during deletion or insertion?

  ➠ Using 2 pointers
- A node in doubly linked list

```
typedef struct node *nodePointer;
typedef struct node {
    nodePointer llink;
    element data;
    nodePointer rlink;
} node;
```

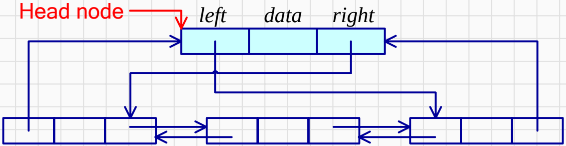


A Head Node

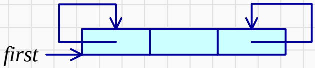
- A head node allows us to implement our operations easily.
  data: no information
- Suppose *ptr* points to any node in a doubly linked list

$$ptr = ptr -> llink -> rlink = ptr -> rlink -> llink$$

- Back or forth with equal ease



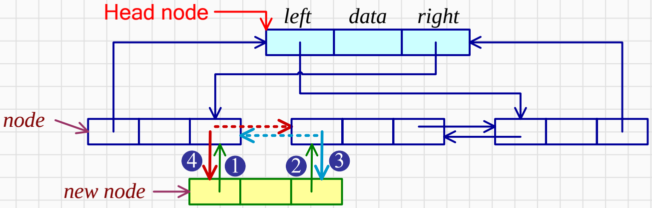
- Emptylist



Empty list

Insertion

```
void dinsert(nodePointer node, nodePointer newnode)
{  /* insert newnode to the right of node
    newnode->llink = node  ❶;
    newnode->rlink = node->rlink  ❷;
    node->rlink->llink = newnode  ❸;
    node->rlink = newnode  ❹;
}
```



Deletion

```
void ddelete(nodePointer node, nodePointer deleted)
{  /* delete from the doubly linked list
   if (node==deleted)
      printf(''Deletion of head node not permitted.  \n'');
   else {
      deleted->llink->rlink = deleted->rlink;  ❶
      deleted->rlink->llink = deleted->llink;  ❷
      free(deleted);
   }
}
```