CSE 326
Lecture 4

Spring Quarter 1995

ADTs that use Simple Linked Lists

- Simple lists - $a_1,a_2,a_3, ..., a_n$

  insert x into k-th position, delete from k-th position, delete a particular item, length
- Ordered lists - $a_1,a_2,a_3, ..., a_n$, where $a_i < a_{i+1}$, for i = 0 to n-1
- Stacks -

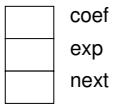
  LIFO - push, pop, top
- Queues -

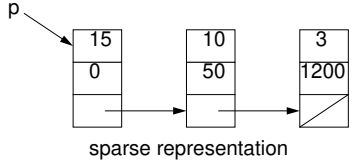
  FIFO - enqueue, dequeue
- Polynomials - $15 + 10x^{50} + 3x^{1200}$

  add, subtract, multiply, divide, mod, evaluate

Polynomials

- $15 + 10x^{50} + 3x^{1200}$

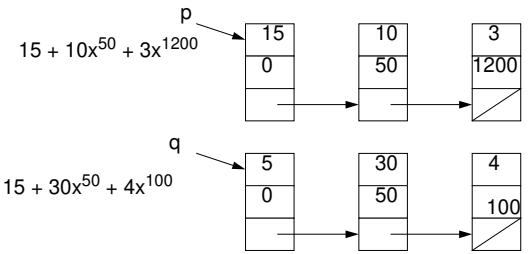




- A dense representation would not have the "exp" field or would be an array of coefficients.
- Sparse represention is very similar to an ordered list and is generally better than a dense represention, because less storage needed for data and less time for operations.
- In the sparse representation the 0 polynomial is represented by null.

Addition of two polynomials

- Similar to merging two sorted lists.
- May be destructive (have side-effects) or non-destructive.



A Simple Recursive Addition (Nondestructive)

- Add(p,q) {

```
local pointer r;
case {
p = null:  r := copy(q);
q = null:  r := copy(p);
p.exp = q.exp: { r := new(node);
                 r.coef = p.coef + q.coef;
                 r.exp = p.exp;
                 r.next := Add(p.next,q.next)};
p.exp < q.exp: {r := new(node);
                 r.coef = p.coef;
                 r.exp = p.exp;
                 r.next := Add(p.next,q)};
q.exp < p.exp: {r := new(node);
                 r.coef = q.coef;
                 r.exp = q.exp;
                 r.next := Add(p,q.next)}}:
return(r)}
```


A Simple Recursive Addition (Destructive)

- Add(p,q) {

```
local pointer r;
case {
p = null:  r := q;
q = null:  r := p;
p.exp = q.exp: {r := p;
                 r.coef = p.coef + q.coef;
                 r.next := Add(p.next,q.next);
                 destroy(q)};
p.exp < q.exp: {r := p;
                 r.next := Add(p.next,q)};
q.exp < p.exp: {r := q;
                 r.next := Add(p,q.next)}}:
return(r)}
```


Headers and Sentinels

- Header to a list:

  - First member of a list may be of a the same or different type as the list nodes.

  - May contain additional fields such as, length of the list, degree of the polynomial, largest coefficient, etc.

  - Very natural in object oriented languages such as C++, Smalltalk,...

- Sentinel to a list

  - First member of a list of the same type as list nodes.

  - Supposed to make programming "easier".

  - Often used in iterative programs.

  - Sentinels can be used in arrays as well as linked lists.

Example of Sentinel

- recursive max without sentinel

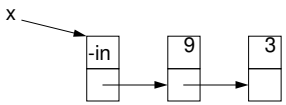
```
max(x) {

local integer m;
if x = null then return(-infinty)
  else  { m := max(x.next);
     if m > x.data then return(m) else return(x.data) }}
```

- iterative max with sentinel containing "-infinity"

```
max(x) {

local integer m;
m := x.data;
while not(x=null) do {
     if x.data > m  then m := x.data;
     x := x.next}
return(m) }
```



Doubly Linked List

- Uses

  - list needs to be traversed in two directions
  - deletion in the middle of a list needs to be efficient

- Example ADT - String Editing

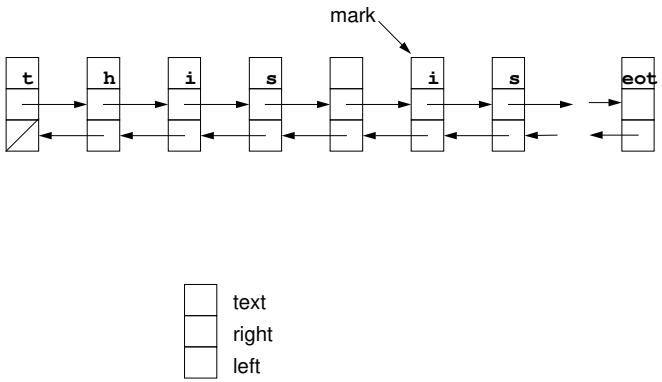
  text string - $a_1a_2a_3...a_n$ with two positions "cursor" and "mark"

```
this is an example of a text string
     ↑                    ↑
    mark                cursor
```

- Some operations

  - Insert(c): inserts the character c just before the cursor.
  - Delete: deletes character just before cursor.
  - Long_Delete: deletes character from cursor to mark
  - Move_Left, Move_Right, Set_Mark, ...

Doubly Linked List for Text String



Example Operations

- Move_Right {

  if not(cursor.text = eot) then cursor := cursor.right}

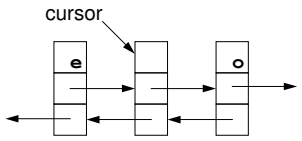
- Move_Left {

  if not (cursor.left = null) then cursor := cursor.left}

- Insert(c) {

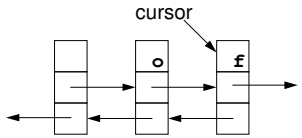
  local pointer p;

```
p := new(node);
p.text := c;
p.right := cursor;
p.left := cursor.left;
cursor.left := p;
if not(p.left = null) then
        p.left.right := p}
```



Delete Operation

- Delete - tread carefully

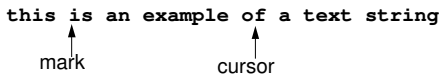


if not(cursor.left = null) then cursor.left := cursor.left.left;

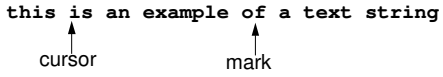
if (not(cursor.left = null) then cursor.left.right = cursor.

Long Delete

- If mark = cursor do nothing
- if mark left of cursor then delete mark to just before cursor



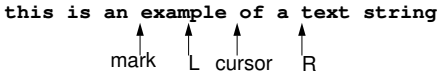
- if mark right of cursor then delete cursor to just before mark, thereby moving cursor to mark.



Testing Order of Cursor and Mark

1. Scan a pointer p from cursor to eot looking for p = mark.

   - may be slow if cursor at beginning of long text.

2. Scan two pointers from cursor, one to the left and one to the right, each one looking for mark.

   - if the cursor and mark are close to each other then this will be much faster.