Linear Lists

Computer Science uses:
- Files on disk/in memory
- Compilers (parsers, arithmetic evaluation, etc.)
- Editors (emacs)
- Operating Systems (Scheduling CPU, Printers, DMA, etc.)
- Databases
- Servers
- Implementing other Data structures
- Basis of functional languages

Sequential Storage

- Items stored in order next to each other
- Inserting into ordered sequential storage list

e.g. insert 6

6

3	5	7	10	12		…

shuffle and copy

3	5	7	7	10	12		…

3	5	6	7	10	12		…

delete similar

Problems?
- O(n) moves on average
- If array full (overflow), change code and recompile

Solutions?
- Linked allocation (but that has problems too)

Linked Storage

- Can be done with parallel arrays or pointers

Parallel arrays

First Available: 0

2

10	5	3	12	7

3	4	1	-1	0

Delete 7, insert 6:

First Available:

Pointers

```
typedef struct NodeBox {
  int info;
struct NodeBox *link;
} NodeType;
```

```
NodeType *list;
list = NULL;
```

*** Example shown in class ***

Insert new node in ordered list

- Issues:
- List empty
- Falling off list end while searching
- First node is special case in code
- Once find spot to insert, have gone too far

Issues addressed by: circular list with header

*** Example shown in class ***

```
insert (item, list) {
  if ((new = GetNewNode())==NULL) overflow();
   else
     new->info = item;
        point = list;
        while(point->link != list && point->link->info < item)
  point = point->link;

     new->link = point-> link;
     point->link = new;
}

delete (list, item) {
   point=list;
   while(point->link != list && point->link->info != item)
     point=point->link;
  if (point->link==list) return -1;//not found
     else {
         temp=point->link;
       point->link = point->link->link;
     free(temp);
  }
     }
```

Used “look ahead”

Time O of Growth

- Typically count number of pointers followed

Insert

Best: insert onto 1st position on list
“while” not done -> O cost -> O(1)

Worst: last position (n+1)
“while” done nx - > O(n)

Average: into position 1 0 cost
into position 2 1 cost

1	0 cost
2	1 cost
3	2 cost
…	
n	n-1 cost
n+1	n cost

avg 0 + 1 +2 + .. + n /(n+1_ = n(n+1)/2(n+1) = 1/2n -> O(n)

2-Way Linked List/Doubly-linked list

circular, header

```
typedef struct NodeTag {
   int info;
  struct NodeTag *left-link;
   struct NodeTag *right-link;

} NodeType;
```

don’t need look ahead

```
insert (item, list) {
  if ((new = GetNewNode())==NULL) overflow();
   else
     new->info = item;                                              A
      point = list->right-link;                                     B
      while(point != list && point->info < item)                    C
         point = point->link;                                       D

     new->left-link = point->left-link;                             EF
     new->right-link = point;                                       G
     point->left_link->right-link = new;                            HI
     point->left_link = new;                                        J

}
```

Time O of G “long” way (assume each pointer reference takes “1” cost)

Position	Number of points	
1	A B C E-J	2 + 1 + 6
2	A B C D C E-J	2 + 3 + 6
3	A B C D C D C E-J	2 + 5 + 6
4	A B C D C D C D C E-J	2 + 7 + 6
…		
n		2 + (2n-1) + 6
n+1		2 + (2(n + 1) -1) + 6

Best: 2 + 1 + 6 -> O(1)

Worst: 2 + (2(n+1)-1)+6 = 2n+9 -> O(n)

Average: (1+8) + (3+8) + (5+8) + … + ((2n-1) + 8) + ((2(n+1)-1)+8) / (n+1)

= (n+1)*8 + 1+3+5+7+…+ (2n-1) + 2(n+1)-1 / (n+1)

= 8(n+1) + (n+1)2 / (n+1) = 8 + n + 1

➔ O(n)

For the “general” way relies on fact that A, B, E-J would get ignored anyway, just counting number of times through “while” loop

When is it more space-efficient to implement a list length n as a 2-way linked list rather than an array of MAX items? (assume items are integers)
Assume integers take I bytes of storage and pointers take P bytes.

Array:	Max*I	- array items
	I	- index of last item (n) or sentinel value
	p	- variable for array itself
	MAXI + I + P	

Linked List:	In	- the n items
	2pn	- 2 pointers for each item
	I+2p	- header
	p	- pointer to (list) i.e. pointer to head
	In+2pn+I+2p+p	

Linked List is better when

$$In + 2pn + I + 2p + p < MAXI + I + p$$
$$n(I+2p) < MAXI+I+P-I-2p-p$$
$$n < (MAXI - 2p)/(I+2p)$$

e.g. when the list has 1000 items, Linked List use less storage when (and pointers & integers each take 4 bytes):

$$1000 < 4MAX - 8 / 12$$
$$3002 < MAX$$

e.g. when MAX is 10,000, what size should list be such that Linked List more space efficient:

$$n < (10000*4 - 8)/12$$
$$n < 3333$$