

Linked List

- Alternate approach to maintaining an array of elements
- Rather than allocating one large group of elements, allocate elements as needed
- Q: how do we know what is part of the array?
A: have the elements keep track of each other
 - use pointers to connect the elements together as a *LIST* of things

Outline

Linked list

basic terms (data, link, node)

type definition

visualization

operations

init, insert, read group, find, print, delete, sort

variations

dummy head node

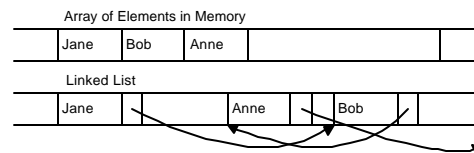
sorted list

Limitation of Arrays

- An array has a limited number of elements
 - routines inserting a new value have to check that there is room
- Can partially solve this problem by reallocating the array as needed (how much memory to add?)
 - adding one element at a time could be costly
 - one approach - double the current size of the array
- A better approach: use a *Linked List*

Dynamically Allocating Elements

- Allocate elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer



Linked List Notes

- Need way to indicate end of list (NULL pointer)
- Need to know where list starts (first element)
- Each element needs pointer to next element (its link)
- Need way to allocate new element (use malloc)
- Need way to return element not needed any more (use free)
- Divide element into data and pointer

Linked List Type

- Type declaration:


```
typedef struct ESTRUCT {
    DataType data; /* DataType is type for element of list */
    struct ESTRUCT *next;
} EStruct, *EPtr;
```

- Pointer to first element:


```
EPtr ListStart;
```

/* ListStart points to first element of list
*ListStart is first element struct
ListStart->data is data of first element
ListStart->next points to second element */

Sample Linked List Operations

```
void main() { /* Assume data is int */
    EPtr ListStart = NULL;
    /* safest to give ListStart an initial legal
       value -- NULL indicates empty list */
```

Pictorially
ListStart
☒

In Memory
ListStart
| 0 |
| 100 |

```
ListStart = (EPtr) malloc(sizeof(EStruct));
/* ListStart points to memory allocated at
   location 108 */
```

ListStart
☐ → [? | ?]
Data Next

ListStart Data Next
| 108 | ? | ? |
| 100 | 108 |

Sample Linked List Ops (cont)

```
ListStart->data = 5;
```

ListStart
☐ → [5 | ?]

ListStart
| 108 | 5 | ? |
| 100 | 108 |

```
ListStart->next = NULL;
```

ListStart
☐ → [5 | ☒]

ListStart
| 108 | 5 | 0 |
| 100 | 108 |

```
ListStart->next = (EPtr) malloc(sizeof(EStruct));
```

ListStart
☐ → [5 | ☐] → [? | ?]

ListStart
| 108 | 5 | 120 | ? | ? |
| 100 | 108 | 120 |

```
ListStart->next->data = 9;
```

ListStart
☐ → [5 | ☐] → [9 | ☒]

ListStart
| 108 | 5 | 120 | 9 | 0 |
| 100 | 108 | 120 |

Sample Linked List Ops (cont)

```
ListStart->next->next = (EPtr) malloc(sizeof(EStruct));
```

ListStart
☐ → [5 | ☐] → [9 | ☐] → [? | ?]

ListStart
| 108 | 5 | 120 | 9 | 132 | ? | ? |
| 100 | 108 | 120 | 132 |

```
ListStart->next->next->data = 6;
```

ListStart
☐ → [5 | ☐] → [9 | ☐] → [6 | ☒]

ListStart
| 108 | 5 | 120 | 9 | 132 | 6 | 0 |
| 100 | 108 | 120 | 132 |

```
/* Linked list of 3 elements (count data values):
   ListStart points to first element
   ListStart->next points to second element
   ListStart->next->next points to third element
   and ListStart->next->next->next is NULL to
   indicate there is no fourth element */
```

Sample Linked List Ops (cont)

```
/* To eliminate element, start with free operation */
free(ListStart->next->next);
```

ListStart
☐ → [5 | ☐] → [9 | ☒]

ListStart
| 108 | 5 | 120 | 9 | 132 | 6 | 0 |
| 100 | 108 | 120 | 132 |

```
/* NOTE: free not enough -- does not change memory
   Element still appears to be in list
   But C might give memory away in next request
   Need to reset the pointer to NULL */
```

```
ListStart->next->next = NULL;
```

ListStart
☐ → [5 | ☐] → [9 | ☒]

ListStart
| 108 | 5 | 120 | 9 | 0 | 6 | 0 |
| 100 | 108 | 120 | 132 |

```
/* Element at 132 no longer part of list (safe to
   reuse memory) */
```

Common Mistakes

- Dereferencing a NULL pointer

```
ListStart = NULL;
ListStart->data = 5; /* ERROR */
```
- Using a freed element

```
free(ListStart->next);
ListStart->next->data = 6; /* PROBLEM */
```
- Using a pointer before set

```
ListStart = (EPtr) malloc(sizeof(EStruct));
ListStart->next->data = 7; /* ERROR */
```

List Initialization

Certain linked list ops (init, insert, etc.) may change element at start of list (what ListStart points at)

- to change what ListStart points to could pass a pointer to ListStart (pointer to pointer)
- alternately, in each such routine, always return a pointer to ListStart and set ListStart to the result of function call (if ListStart doesn't change it doesn't hurt)

```
EPtr initList() {
    return NULL;
}

ListStart = initList();
```

A Helper Function

Build a function used whenever a new element is needed (function always sets data, next fields):

```
EPtr newElement(DataType ndata, EPtr nnext) {
    EPtr newEl = (EPtr) malloc(sizeof(EStruct));
    newEl->data = ndata;
    newEl->next = nnext;
    return newEl;
}
```

List Insertion (at front)

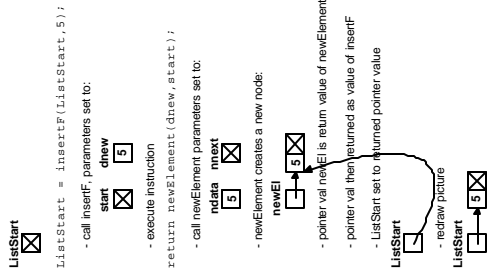
- Add new element to list:
 - Make new element's next pointer point to start of list
 - Make pointer to new element start of list

```
EPtr insertF(EPtr start, DataType dnew) {
    return newElement(dnew, start);
}
```

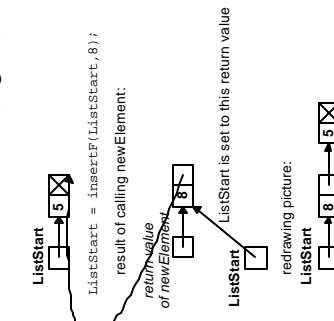
– To use, get new value to add (ask user, read from file, whatever), then call insertF:

```
ListStart = insertF(ListStart, NewDataValue);
```

Insert at Front Example



Insert at Front (Again)

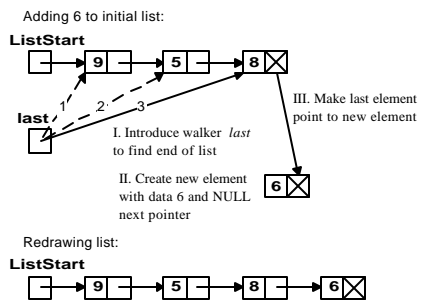


Insert at End

Need to find end of list -- use *walker* (temporary pointer) to "walk" down list

```
EPtr insertE(EPtr start, DataType dnew) {
    EPtr last = start; /* Walker */
    if (start == NULL) /* if list empty, add at */
        return newElement(dnew, NULL); /* start */
    else {
        while (last->next != NULL) /* stop at */
            last = last->next; /* last item */
        last->next = newElement(dnew, NULL);
        return start; /* start doesn't change */
    }
}
```

Insert at End Example



Reading a Group of Elements

```

EPtr readGroup(EPtr start, FILE *instream) {
    DataType dataitem;
    while (readDataSucceeds(instream,&dataitem))
        /* Add new item at beginning */
        start = newElement(dataitem,start);
    return start;
}

/* Assume DataType is int: */
int readDataSucceeds(FILE *stream, int *data) {
    if (fscanf(stream,"%d",data) == 1)
        return 1;
    else
        return 0;
}

```

Reading Group - Add at End

```

EPtr readGroupE(EPtr start, FILE *instream) {
    EPtr last;
    DataType data;

    /* Try to get first new item */
    if (!readDataSucceeds(instream,&data))
        return start; /* if none, return initial list */
    else { /* Add first new item */
        if (start == NULL) {
            /* If list empty, first item is list start */
            start = newElement(data,NULL);
            last = start;
        }
    }
}

```

Reading Group - Add at End (cont)

```

else { /* List not initially empty */
    last = start;
    while (last->next != NULL) /* Find end */
        last = last->next;
    /* Add first element at end */
    last->next = newElement(data,NULL);
    last = last->next;
}

/* Add remaining elements */
while (readDataSucceeds(instream,&data)) {
    last->next = newElement(data,NULL);
    last = last->next;
}
return start;
}

```

Printing a List

Use a walker to examine list from first to last

```

void printList(EPtr start) {
    EPtr temp = start;

    while (temp != NULL) {
        printData(temp->data);
        temp = temp->next;
    }
}

```

Finding an Element in List

- Return pointer to item (if found) or NULL (not found)

```

EPtr findE(EPtr start, DataType findI) {
    EPtr findP = start; /* walker */

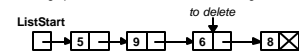
    while ((findP != NULL) &&
           (findP->data is not the same as findI))
        findP = findP->next;

    return findP;
}

```

Deleting an Element

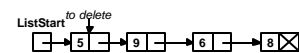
Need to change pointer of record before the one being deleted



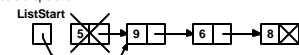
delete operation:



But what if NO element before the one to be deleted?



delete operation:



Deletion Code

```
EPtr delete(EPtr start, DataType delI) {
    EPtr prev = NULL;
    EPtr curr = start;
    while ((curr != NULL) && (curr->data is not delI)) {
        prev = curr;
        curr = curr->next;
    }
    if (curr == NULL)
        printf("Item to delete not found\n");
    else {
        if (prev == NULL)
            start = start->next;
        else
            prev->next = curr->next;
        free(curr);
    }
    return start;
}
```

Selection Sorting Linked List

```
EPtr sort(EPtr unsorted) {
    EPtr sorted = NULL;
    EPtr largest, prev;
    while (unsorted != NULL) {
        prev = findItemBeforeLargest(unsorted);
        if (prev == NULL) {
            largest = unsorted;
            unsorted = unsorted->next;
        }
        else {
            largest = prev->next;
            prev->next = largest->next;
        }
        largest->next = sorted;
        sorted = largest;
    }
    return sorted;
}
```

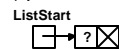
Item Before Largest

```
EPtr findItemBeforeLargest(EPtr start) {
    EPtr prevlargest = NULL;
    EPtr largest = start;
    EPtr prev = start;
    EPtr curr = start->next;
    while (curr != NULL) {
        if (curr->data bigger than largest->data) {
            prevlargest = prev;
            largest = curr;
        }
        prev = curr;
        curr = curr->next;
    }
    return prevlargest;
}
```

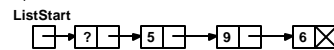
Linked List Variation: Dummy Head Node

Using "dummy" first (head) node:

Empty linked list



Sample list:



- Why?
 - No special case for inserting/deleting at beginning
 - ListStart does not change after it is initialized
- Disadvantage
 - cost of one extra element

DH List Initialization/Insert

```
EPtr initList() {
    EPtr ListStart =
        (EPtr) malloc(sizeof(EStruct));
    ListStart->next = NULL;
    return ListStart;
}

void insertF(EPtr start, DataType dnew) {
    start->next =
        newElement(dnew, start->next);
}
```

DH Inserting at End

```
void insertE(EPtr start, DataType dnew) {
    EPtr last = start; /* Walker */
    /* No special list is empty case */
    while (last->next != NULL)
        last = last->next;
    last->next = newElement(dnew, NULL);
}
```

DH Printing a List

Have walker start at second element

```
void printList(EPtr start) {
    EPtr temp = start->next;

    while (temp != NULL) {
        printData(temp->data);
        temp = temp->next;
    }
}
```

DH Deletion Code

```
void delete(EPtr start, DataType delI) {
    EPtr prev = start;
    EPtr curr = start->next;
    while ((curr != NULL) && (curr->data != delI)) {
        prev = curr;
        curr = curr->next;
    }
    if (curr == NULL)
        printf("Item to delete not found\n");
    else {
        prev->next = curr->next;
        free(curr);
    }
}
```

Linked List Variation: Sorted List

Idea: keep the items on the list in a sorted order

sort based on data value in each node

advantages:

- already sorted
- operations such as delete, find, etc. need not search to the end of the list if the item is not in list

disadvantages

- insert must search for the right place to add element (slower than simply adding at beginning)

Sorted Insert (with Dummy Head)

```
void insertS(EPtr start, DataType dnew) {
    EPtr prev = start;
    EPtr curr = start->next;
    while ((curr != NULL) &&
           (dnew < curr->data)) {
        prev = curr;
        curr = curr->next;
    }
    prev->next = newElement(dnew, curr);
}
```