

Subtopic: Linked lists

- A linear structure, like an array
- Composed of self-referential *nodes*
- Operations: insert, delete, traverse
- Array implementation
- Dynamic-allocation implementation

David Keil 1/03 1

Inserting into a sorted collection

- Array or random-access file may require many steps to open a slot
- Insertion into a *chain* takes 3 steps
- Assumes knowledge of *where* to insert

David Keil 1/03 2

A linked list

- Linked lists are chains of *nodes*
- Their form is more flexible than the cell structure of an array or random-access file

David Keil 1/03 3

About linked lists

- A second way to implement collections
- Contain self-referential node structures
- Operations:
 - Insertion
 - Deletion
 - Traversal
- Array implementation
- Dynamically -allocated node implementation

David Keil 1/03 4

List-node structure type

```

struct list_nodes
{
    item_type data;
    node_references next;
};
    
```

...where

- $\langle node_references \rangle$ is some data type (e.g., subscript or pointer) that refers to a node
- $\langle item_type \rangle$ is the data type of the value stored by a node

David Keil 1/03 5

Array implementation of a linked list

- The *next* link is an integer subscript
- A special subscript value (-1 here) denotes end of list

David Keil 1/03 6

C++ array-based linked list

```

const int MAX_RECS = 4, MAX_NAME_SIZE = 9;
struct nodes
{
    char name[MAX_NAME_SIZE];
    int next;
};
class arr_lists
{
public:
    arr_lists();
    void read();
    void show();
private:
    int predecessor(char *value);
    void insert_node(char *value);
};
    arr_lists:arr_lists()
    // Default constructor.
    {
        num_recs = 0;
        for (int i=0; i < MAX_RECS; ++i)
        {
            node[i].name[0] = '\0';
            node[i].next = -1;
        }
        header.next = -1;
    }
    [arrlist.cpp]

```

David Keil 1/03 7

List traversal and insertion

```

void arr_lists::show()
// Displays list in alphabetical order.
{
    int i = header.next;
    while (i != -1)
    {
        cout << node[i].name << endl;
        i = node[i].next;
    }
}

```

- To prepend a node, use next available array element (subscript *num_recs*)
- Assign its *next* the value of *header.next*; then make *header.next* the node's subscript

David Keil 1/03 8

Array-implementation code

```

struct nodes
{
    char name[20];
    int next;
};
void main()
{
    nodes list[4], header = {"", -1};
    list[0] = {"Ed", -1};
    header.next = 0;
    list[3] = {"Ana", -1};
    list[3].next = 0;
    header.next = 3;
}

```

Node has a subscript as its link to next node

Initialize empty list

Insert a node

Insert another

David Keil 1/03 9

Linked-list insertion algorithm

List-insert (target, value)

> Inserts node storing *value* into *list* after node referred to as *target*

- Create new node using *value*
- Let *p* point to new node
- next(p) → next(target)*
- next(target) → p*

David Keil 1/03 10

Node insertion by prepend

```

struct nodes
{
    char name[20];
    nodes* next;
};
struct lists
{
    nodes header;
};
void list_prepend(lists* lst, char *s)
{
    nodes* new_node = new nodes;
    strcpy(new_node->name, s);
    new_node->next = lst->header.next;
    lst->header.next = new_node;
}

```

- List_prepend* inserts a node containing data value *s* at the beginning of linked list *lst*.

David Keil 1/03 11

Inserting into a sorted list

```

void list_insert_in_order(lists* lst, char *s)
// Inserts string <s> into <lst> in alphabetical order
{
    nodes *p1 = &(lst->header), *p2 = p1->next;
    *new_node = (nodes*)malloc(sizeof nodes);
    bool done = false;
    // Preconditions: <lst> points to header node, s != NULL
    strcpy(new_node->name, s);
    while(p1 != NULL && !done)
    {
        // Loop invariant: all nodes up to p1 have values < <s>
        if (p2 == NULL || strcmp(p2->name, s) > 0)
        {
            new_node->next = p2;
            p1->next = new_node;
            done = true;
        }
        p1 = p2;
        p2 = p2->next;
    }
    // Postcondition: <lst> contains <s> and is in alphabetical order
}

```

David Keil test 12

List-traversal algorithm

```

Iterator ← next(header);
While iterator not null
    Visit value(iterator)
    Iterator ← next(iterator)
    
```

- “Visit” is whatever operation is to be carried out on each node of the list; e.g., display data value

David Keil 1/03 13

List traversal in C

```

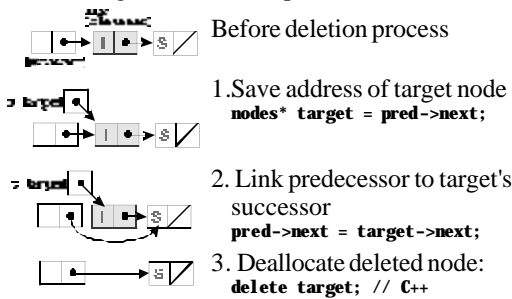
void list_display(lists lst)
/* Outputs data values of all nodes of
list <lst>. */
{
    /* Counter is stored in pointer: */
    nodes* p_node = lst.header.next;

    /* Loop through list, displaying data: Iterator
while (p_node != NULL)
    {
        printf("%s ", p_node->value);
        p_node = p_node->next;
    }
}
    
```

David Keil 1/03 14

Linked-list node deletion

(given address of predecessor)



David Keil 1/03 15

C++ code to delete a node

```

void list_delete_node(nodes* pred)
// Deletes the node after <pred> from list.
{
    /* No action on null parm or null
successor to it: */
    if (pred == NULL) return;
    if (pred->next == NULL) return;

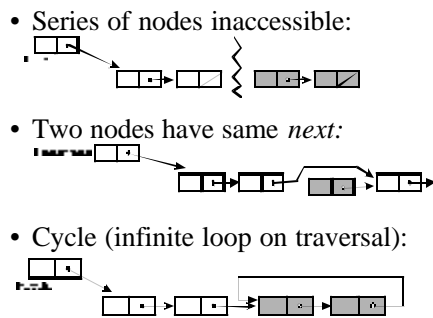
    /* Link predecessor of target to
<pred>'s successor: */
    nodes* target = pred->next;
    pred->next = target->next;

    /* Deallocated deleted node: */
    delete target;
}
    
```

C++ operator; in C use function free

David Keil 1/03 16

3 situations to avoid



David Keil 1/03 17

C++ class for list nodes

```

class list_nodes
{
public:
    list_nodes() { *value = next = NULL; };
    list_nodes(char *s)
        { strcpy(value, s); next = NULL; };
    char *get_value() { return value; };
    list_nodes *get_next() { return next; };
private:
    char value[80];
    list_nodes *next;
};
    
```

David Keil 1/03 18

C++ class for linked list

```

class linked_lists
{
public:
    linked_lists() { header.next = NULL; };
    void prepend(char *s);
    void append(char* s);
    nodes* search(char* s);
    void delete_node(list_nodes *pred);
    void display();
    list_nodes *first() { return header.next; };
    list_nodes *get_header() { return &header; };
private:
    list_nodes header;
};
    
```

David Keil 1/03 19

Inserting and deleting with list of strings

```

void main()
{
    linked_lists list;
    char input[80];
    cout << endl << endl;
    do {
1      cout << "Enter a string (* to quit): ";
        cin >> input;
        if (strcmp(input, "") != 0)
            list.prepend(input);
    } while (strcmp(input, "") != 0);
    list.display();
2  list_nodes *p_node = list.first(), *p_prev = list.get_header();
    while (p_node != NULL) {
3      cout << "Delete " << p_node->get_value() << "?\n";
        if (toupper(_getch()) == 'Y')
            list.delete_node(p_prev);
        else
            p_prev = p_node;
        p_node = p_prev->get_next();
    }
    cout << "\nWhat remains is: \n";
    list.display();
4 }
    
```

This program
 (1) builds a linked list of strings from user input,
 (2) displays it in reverse order,
 (3) prompts for deletions, and
 (4) displays result.

[strlist.cpp]
David Keil 1/03 20

Prepend for linked list of strings

```

void linked_lists::prepend(char *s)
/* Inserts a node containing value <s>
into linked list. */
{
    list_nodes *new_node =
        new list_nodes(s);
    if (header.next != NULL)
        new_node->next = header.next;
    header.next = new_node;
}
    
```

[strlist.cpp] David Keil 1/03 21

Delete for linked list of strings

```

void linked_lists::delete_node(list_nodes *pred)
// Deletes the node following <pred> from list.
// Precondition: <pred> points to a node or NULL
{
    // Take no action on null parameter:
    if (pred != NULL)
    {
        // Record address of node to be deleted:
        list_nodes *deleted_node = pred->next;
        // Link predecessor of deleted node to
        // its successor:
        pred->next = deleted_node->next;
        delete deleted_node;
        // Postcondition: list containing <pred>
        // no longer contains <pred>'s successor.
    }
}
    
```

David Keil 1/03 22

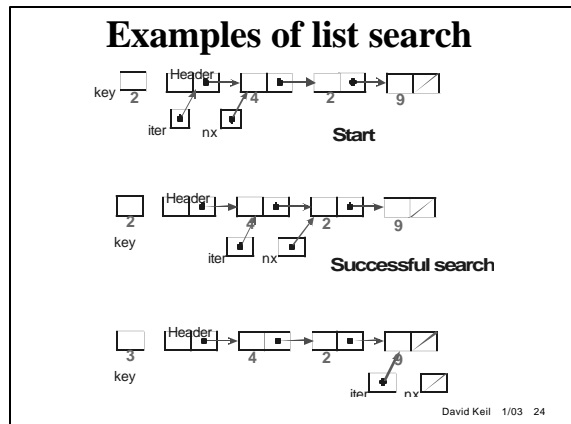
Searching a linked list of int

```

struct nodes { int data; nodes* next; };
struct lists { nodes hdr; nodes* search(int); };

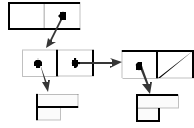
nodes* lists::search(int key)
// Returns pointer to predecessor of node
// with value <key>, or NULL on failure.
{
    nodes *iter = &hdr, *nx = hdr.next;
    while (nx != NULL)
    {
        // Loop invariants: 1. <key> not found yet;
        // 2. <nx> points to node after <iter>.
        if (key == nx->data)
            return iter;
        iter = nx;
        nx = nx->next;
    }
    return NULL;
    // Postcondition: either pointer to predecessor of node
    // containing <key> is returned, or <key> not in list
    // and search returns NULL.
}
    
```

[to test] David Keil 1/03 23



Generic linked lists

- If the *data* member of a node can be of any type, the collection is general-purpose



- One way to do this: use *void* pointers
- Another way, in C++: class template, e.g., as done in the Standard Template Library (STL)

David Keil 1/03 25