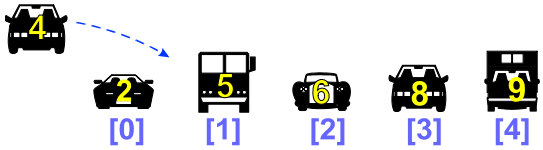
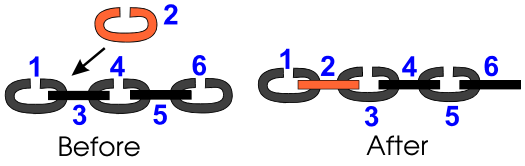
Linked lists

- Data structures with linked components
- *Operations:* insertion, search, deletion, traversal
- Doubly linked lists

Inserting into a sorted collection

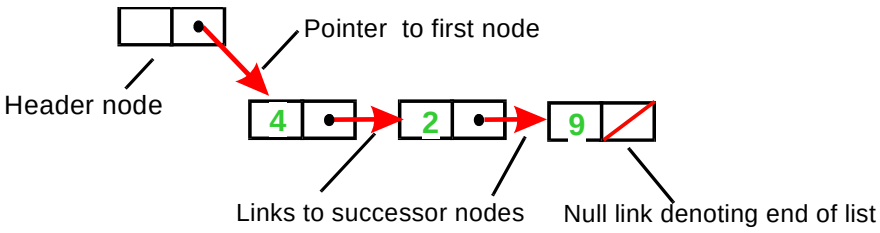


- Array or random-access file may require many steps to open a slot



- Insertion into a *chain* takes 3 steps
- Assumes knowledge of *where* to insert

A linked list



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

David Keil 9/04 3

About linked lists

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

David Keil 9/04 4

List-node structure type

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

…*where*

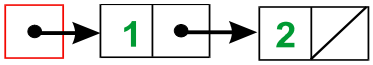
- ⟨*node_references*⟩ is some data type (e.g., subscript or pointer) that refers to a node
- ⟨*item_type*⟩ is the data type of the value stored by a node

David Keil 9/04 5

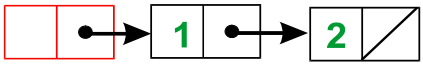
Linked lists with dynamically allocated nodes

3 implementations; "list" item is:

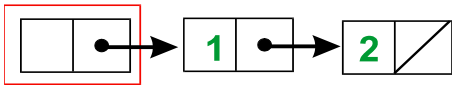
- pointer to first node;



- header node;



- object containing header node



David Keil 9/04 6

Initialize node

1. Assign new value to data member
2. Assign NULL to link member

Prepend node

1. Link new node to *next* of header
2. Link header to new node

Append node

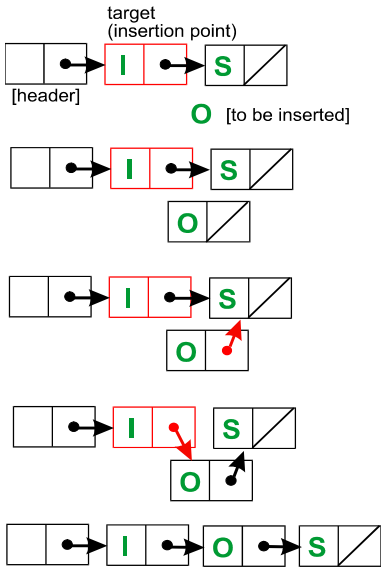
1. Step to end of list
2. Link last node to new node

Linked-list node insertion

Before insertion

1. Allocate new node to store new value
2. Link new node to successor
3. Link target to new node

After insertion



Node insertion by prepend (C++)

```
struct nodes
{
  int value;
  nodes* next;
};
```

```
class lists
{
public:
  prepend(int v);
private:
  nodes header;
};
```

```
void lists::prepend(int v)
// Inserts node w/<v> at front.
{
  nodes* p_new_node = new nodes;
  p_new_node->value = v;
  p_new_node->next = header.next;
  header.next = p_new_node;
}
```


List-search (*hdr, key*)

[Preconditions: *hdr* is pointer to header node of linked list, *key* is value to be found]

i ← *next* (*hdr*)
while *i* ≠ NULL
  if *value*(*i*) = *key*
    return *i*
  else
    i ← *next*(*i*)
return NULL

[Postcondition: Pointer to node containing *key* is returned, or NULL if *key* not found.]

Using *strcmp* to compare strings

```
struct nodes
{
  char name[40];
  nodes* next;
};
void main()
{
  nodes* p_node = new nodes;
  strcpy(p_node->name,”Bob”);
  cout << "Search key: ";
  char key[40];
  cin >> key;
  if (strcmp(key,p_node->name) == 0)
    cout << "Match" << endl;
}
```

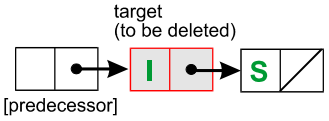

C code for list search

(assumes *nodes* structure type)

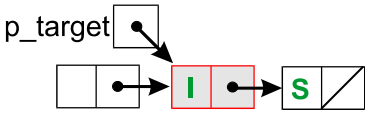
```
nodes* search_list(nodes hdr,char* key)
/* Returns pointer to node containing
   <key>, as value of its member <name>,
   in list headed by <hdr>, or NULL if
   <key> not found. */
{
   nodes* p = hdr.next;
   while (p != NULL) {
     if (strcmp(p->name,key) == 0)
       return p;
     else
       p = p->next;  }
   return NULL;
}
```

Linked-list node deletion

(given address of predecessor)

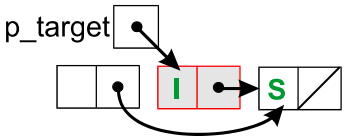


Before deletion process



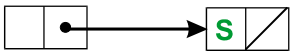
1. Save address of target node

```
nodes* target = pred->next;
```



2. Link predecessor to target's successor

```
pred->next = target->next;
```



3. Deallocate deleted node:

```
delete target; // C++
```


C code to delete a node

```
void list_delete_node(nodes* pred)
// Deletes the node after <pred> from list.
{
  nodes* target;

  /* No action if parm or successor null: */
  if (pred == NULL)     return;
  if (pred->next == NULL)    return;

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

  /* Deallocated deleted node: */
  free(target);
}
```

List traversal to display (C)

```
void list_display(lists lst)
/* Outputs values in all nodes of <lst>. */
{
  /* Iterator is a pointer: */
  nodes* p_node = lst.header.next;

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


A linked list of strings (C++)

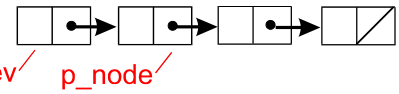
```
void linked_lists::prepend(char *s)
// Inserts a node containing value <s> at start of
list.
{
  list_nodes *new_node = new list_nodes(s);
  new_node->next = header.next;
  header.next = new_node;
}
void linked_lists::delete_node(list_nodes *pred)
// Deletes the node following <pred> from list.
{
  // Take no action on null parameter:
  if (! pred || ! pred->next) return;
  // 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;
}
```

[*strlist.cpp*]

Inserting, deleting with a list of strings

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

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



[*strlist.cpp*]

A collection class using a linked list

```
class list_nodes
{
public:
   list_nodes() { next = NULL; };
   list_nodes(employees e) { emp = e; next = NULL; };
   employees get_value() const { return emp; };
   list_nodes *get_next() { return next; };
   friend class employee_lists;
private:
   employees emp;
   list_nodes *next;
};

class employee_lists
{
public:
   employee_lists() { header.next = NULL; };
   bool retrieve(char* file_name);
   void prepend(employees e);
   void display();
   list_nodes *first() { return header.next; };
   list_nodes *get_header() { return &header; };
private:
   list_nodes header;
};
```

Node type; instances allocated on heap

An instance of employee_lists *is a collection of* employees

Container class

Challenge problem:

- Can you delete a node of a singly linked list knowing only its address, not its predecessor's?
- *Hint:* You can assign new values to *next* member; what other assignments are possible?

David Keil 9/04 19

Doubly-linked lists

- Each node has link to predecessor as well as successor
- Advantages:
  - Can traverse list backwards and forward
  - Can append a node more quickly
  - Can delete a node knowing only its address
- Cost: one extra pointer per node; one extra step for *each* node in *any* operation
- Is DLL faster or slower in any operation?

David Keil 9/04 20

Threaded lists

- Each list node may have a pointer to a node far ahead in a list; say, 100 nodes ahead
- Thus we can look far ahead without stepping through all nodes
- This restores some of the benefits of binary search of arrays
- Maintenance cost for this arrangement may be high during update operations

David Keil 9/04 21

Pros and cons of linked lists

- Size is limited only by physical memory
- Expansion is smooth; no copying of entire collections to insert one element

- Must visit (n – 1) nodes to access the n^{th} node
- Less than optimal search time
- Overhead: one or two pointers, one or two dereferences, per node

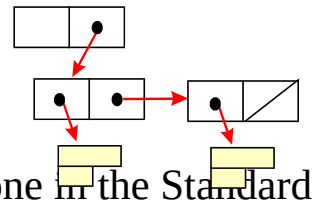
David Keil 9/04 22

Linked-list implementations of collections

- Possible implementations of a node that stores an item *x*:
  - Class with all data attributes of *x*, and *next*
  - Class with members *x*, *next*
  - Generic class with *void* pointer to *x*; *next*
  - Class template (for generic linked list)
- Implementations of collection:
  - Header node
  - Pointer to first node
  - Either implementation may use instantiation of node template as data member

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)



- Example:

```
#include <list>
…
list<employees> roster;
```

A linked-list class template

```
template <class T> class list_nodes
{
public:
   list_nodes() { next = NULL; };
   dlist_nodes(T v) { value = v; next = NULL; };
   T get_value() { return value; };
   void set_next(list_nodes<T> *p) { next = p; };
   list_nodes<T> *get_next() { return next; };
private:
   T value;
   list_nodes<T> *next;
};

template <class T> class linked_lists
{
public:
   linked_lists() { header.set_next(NULL); };
   void append(T v);
   void display();
   list_nodes<T> *first() { return header.get_next(); };
   list_nodes<T> *get_header() { return &header; };
private:
   list_nodes<T> header;
};
```

[*listtplt.cpp*]

2 instantiations of a list template

```
void main()
{
   linked_lists<int> some_numbers;
   int input;
   do {
      cout << "Enter an integer (0 to quit): ";
      cin >> input;
      if (input != 0)
         some_numbers.append(input);
   } while (input != 0);
   some_numbers.display();

   linked_lists<char> word;
   char ch;
   do {
      cout << "Enter a character (\".\" to quit): ";
      cin >> ch;
      if (ch != '.')
         word.append(ch);
   } while (ch != '.');
   word.display();
}
```

list of integers

list of characters

[*listtplt.cpp*]