Computer Science II — CSci 1200 — Sections 1-4,6
Week 6, Friday Class — October 5, 2001
Linked Lists, Part 2

Announcements

• Tests will be returned next Friday. We will discuss test content, test preparations, and grades then.

• No class next Tuesday.

• No labs next week.

• Pick up HW 4 today.

Today’s Class

• Review of linked lists: copying a list.

• Generalizations:
  – Dummy header nodes
  – Circularly linked-lists
  – Doubly-linked lists

• Merge functions
Review: Copying A List

A significant fraction of students had difficulty with the copy_list problem, the last checkpoint in the lab. Here is the solution, which we will go over in class.

```cpp
void
Set0fInt :: copy_list( const Set0fInt& old )
{
    if ( old.head_ ) // the old list is not empty
    {
        head_ = new Set0fIntNode( old.head_->element );
        Set0fIntNode * prev = head_;
        Set0fIntNode * old_curr = old.head_->next;
        while ( old_curr )
        {
            Set0fIntNode * new_ptr = new Set0fIntNode( old_curr->element );
            prev->next = new_ptr;
            prev = new_ptr;
            old_curr = old_curr->next;
        }
    }
    else
    {
        head_ = 0;
    }
}
```
Linked-Lists with Dummy Head Nodes

- Some linked list code is easier if you use a “dummy” head node.
- For example, the empty list looks like

and a SetOfInt list with values 1, 3 and 9 looks like:

- Special-case checks for an empty list are no longer needed
- The constructors and copy_list function must create a dummy head node and have head_point to it.
- Several functions can be simplified. We will look at SetOfInt :: insert in class.
Circularly Linked-Lists with Dummy Head Nodes

- The next step is to make the list circularly linked.

- This doesn’t provide as much as the dummy head node, but it helps in the next step (below), which is to add double linking.

- The empty list now looks like

```
head
```

and a SetOfInt list with values 1, 3 and 9 looks like:

```
head
  DUMMY
  next

  1
  next

  3
  next

  9
```

- The end of list test checks to see if the (dummy) head node has been reached. For example, the following adds the values in the set:

```c
int sum;
for ( SetOfIntNode * p = head_ -> next; p != head_; p = p -> next )
  sum += p -> element;
```

Note the termination check is `p != head_`, in other words, `p` doesn’t point at the dummy head node (where `head_` points).

- The constructors and `copy_list` function must create a dummy head node, have `head_` point to it, and then make it point back to itself.
Doubly-linked, Circular Lists with Dummy Head Nodes

- Now we can add bidirectional linking to the nodes. Each node has a pointer to its predecessor and its successor. These pointers are often called `prev` and `next`.

- Here's a picture of the doubly-linked, circular list with a dummy header containing the values 1, 3 and 9:

- We can now step through the linked list going either forward or backward. Here's how to go backward and add up the values:

```cpp
int sum = 0;
SetOfIntNode * curr = head_ -> prev; // last node
while ( curr != head_ ) // curr is not pointing the same place as head_
{
    sum += curr -> element;
    curr = curr -> prev;
}
```

Of course the while loop could have been a for loop just as well.

- Here's what the body of the default constructor might look like for an empty list:

```cpp
head_ = new SetOfIntNode;
head_ -> next = head_; // point ahead to itself
head_ -> prev = head_; // point back to itself
```

The resulting list looks like:
• Appending a node to the end of the list (with value \( x \)) works quite simply

```cpp
SetOfIntNode * newp = new SetOfIntNode(x);
newp -> next = head_;  
newp -> prev = head_ -> prev;
head_ -> prev -> next = newp;
head_ -> prev = newp;
```

• We can delete a node (except the head node, which we never delete) just by knowing a pointer to it. Suppose this pointer is \( p \). Then the following deletes the node pointed to by \( p \) from the linked list:

```cpp
p -> next -> prev = p -> prev;
p -> prev -> next = p -> next;
delete p;
```

• Special-case checks for an empty list are no longer needed

• Inserting a new node after a given node requires 4 pointer manipulations. We will not discuss this in class because it will be part of your homework.
Merging Ordered Lists

- By making the linked list ordered by increasing value, we can write very efficient set intersection and set union functions.
- These are examples of “merging” algorithms. We consider intersection here.
- Intuition (1): Consider the first element in each set.
  - If the two first elements are equal, then that element can be added to the intersection.
  - Otherwise, the smaller of the two elements can not be in the intersection.
  - Hence, we can consider the next element in the list containing the smaller element.
- Intuition (2): considering the next element in this way gives us a means of stepping through the list.
  - We maintain two pointers, one each to the “current” element of each list.
  - When the current elements are the same (and added to the intersection), both pointers can be moved to the next element.
  - Otherwise the pointer to the smaller element is moved.
- The process ends when one or both of the lists is exhausted.
- Here is working code that implements the idea. It is much more efficient than the version given in SetOfInt.cpp

```cpp
SetOfInt set_intersect( const SetOfInt& s, const SetOfInt& t )
{
    SetOfInt u; // the new set

    // Pointers into the linked list of the old sets

    SetOfIntNode *s_curr = s.head_, *t_curr = t.head_;

    // Pointer to the end of the newly created linked list in the // the new set.
```
SetOfIntNode *u_end = 0;

// Step through the old sets. s_curr and t_curr will always point to
// the first element not yet considered in each set. If the two
// first elements are equal, then add this element to the end of
// the new list. Otherwise, skip over the smaller of the two
// elements in the old lists.

while ( s_curr && t_curr )
{
    if ( s_curr -> element == t_curr -> element )
    {
        SetOfIntNode * node_ptr = new SetOfIntNode( s_curr -> element );

        // Attach to the new list. If this list is currently
        // empty, make the head pointer point to this node.
        if ( !u.head_ )
            u.head_ = node_ptr;
        else
            u_end -> next = node_ptr;
        u_end = node_ptr;

        s_curr = s_curr -> next;
        t_curr = t_curr -> next;
    }
    else if ( s_curr -> element < t_curr -> element )
        s_curr = s_curr -> next;
    else
        t_curr = t_curr -> next;
}
return u;

Summary

• Make sure you understand the process of copying a list.

• Variations on linked lists, such as dummy header nodes, circular linking, and double linking make many linked-list operations easier. The cost is additional space, which may or may not be important (usually
not) depending on the operation.

- Merging ordered lists is a fast operation. This is a general technique used in a number of algorithms.