

## Fun With Linked Lists

---

This handout was written by Julie Zelenski and Jerry Cain.

This material doesn't appear standalone anywhere in the reader, and linked lists are important enough as data structures that we should be discussing them as legitimate top-level containers. Eventually we'll see that linked lists (and more generically, linked structures) are used to back the **Queue**, the **Map**, and the **Set**. But linked lists exist outside the container framework, so it's important we understand the mechanics involved in building, iterating over, and otherwise manipulating them.

### Simple Linked Lists

First, here's our node definition:

```
struct addressRec {
    string name;
    string address;
    string phone;
    addressRec *next;
};
```

Here are the basic operations for creating and printing a single **addressRec** cell:

```
addressRec *getNewAddress() {
    cout << "Enter name (or return to quit): ";
    getline(cin, name);
    if (name == "") return NULL;
    addressRec *newOne = new addressRec;
    newOne->name = name;
    cout << "Enter address: ";
    getline(cin, newOne->address);
    cout << "Enter phone: ";
    getline(cin, newOne->phone);
    newOne->next = NULL;    // initialize field to show no one follows
    return newOne;
}

void printAddress(addressRec *person) {
    cout << person->name << endl;
    cout << person->address << endl;
    cout << person->phone << endl;
}
```

Let's start with the first (unsorted) version of the linked address list. In creating the list, we prepend each new entry to the front of the list, since that's the easiest.

```

addressRec *buildAddressBook() {
    addressRec *list = NULL;
    while (true) {
        addressRec *newOne = getNewAddress();
        if (newOne == NULL) return list;
        newOne->next = list; // attach rest of list to new cell
        list = newOne;      // new cell becomes the head of list
    }
}

```

Note that in freeing the list we have to be careful to not access the current cell after we have freed it.

```

void freeAddressBook(addressRec *list) {
    while (list != NULL) {
        addressRec *next = list->next; // save next ptr before we free this one
        delete list;
        list = next;
    }
}

```

We can use the idiomatic **for** loop, linked list traversal to print each address cell:

```

void printAddressBook(addressRec *list) {
    for (addressRec *cur = list; cur != NULL; cur = cur->next)
        printAddress(cur);
}

```

A linear search is used to look up an entry by name:

```

addressRec *findPerson(addressRec *list, string name) {
    for (addressRec *cur = list; cur != NULL; cur = cur->next)
        if (name == cur->name)
            return cur;

    return NULL;
}

```

Now, we re-consider our decision to keep the list unordered and decide instead to maintain the list in alphabetical order. With this change, we can re-write our **findPerson** function to be a bit smarter. We use **string::compare** to determine if we have exactly matched on the current cell. We can also note if our name has been passed alphabetically, indicating that it ain't in the list, allowing us to bail early:

```

addressRec *findPerson(addressRec *list, string name) {
    for (addressRec *cur = list; cur != NULL; cur = cur->next) {
        int cmp = name.compare(cur->name);
        if (cmp == 0) return cur; // exact match right here
        if (cmp < 0) return NULL; // passed position & didn't find it
    }

    return NULL; // finished off list and didn't find it
}

```

Now, here comes the tricky part. We need a function that'll build the list up in sorted order. The simple add-in-front approach won't work here, as we need to splice the cell somewhere into the middle of the list. The loop we wrote for **findPerson** can find the appropriate position for us, but it'll go one beyond where we need to stop. ☹ After the loop, **cur** points to the cell that will follow **newOne** in the list. Given the forward-chaining properties of linked lists, we have no easy way to get to the cell previous to the one identified by **findPerson**.

How about this? We'll maintain two pointers while walking down the list, using the second pointer to track the previous cell. After the loop, **prev** will point to the cell before the **newOne**, **cur** will point to the cell after. We need to splice **newOne** right in between these two. This means attaching **cur** to follow the new cell and attaching the new cell to follow **prev**. It is possible that the previous cell is **NULL** (when **newOne** is inserted at the head of the list), and thus we must handle this case specially. Since this will require changing list, we need to pass the head pointer by reference, necessitating the **addressRec \*&!**

```

void insert(addressRec*& list, addressRec *newOne) {
    addressRec *cur; // needs to live beyond for loop, so declare here
    addressRec *prev = NULL; // first cell has no previous cell
    for (cur = list; cur != NULL; cur = cur->next) {
        if (newOne->name < cur->name) break; // found place!
        prev = cur;
    }
    // now, "prev" points to one before newOne, "next" is right after
    newOne->next = cur; // works even if cur is NULL
    if (prev != NULL)
        prev->next = newOne;
    else
        list = newOne; // note the special case!
}

```

Now's a good time to think through the special cases and make sure the code handles them correctly. What happens if the cell is being inserted at the very end of the list? What about at the very beginning? What if the list is entirely empty before we start?

Here's how we would call the insertion function to build a sorted address book:

```
addressRec *buildSortedBook() {
    addressRec *list = NULL;
    while (true) {
        addressRec newOne = getNewAddress();
        if (newOne == NULL) break;
        insert(list, newOne); // note list is passed by reference!
    }

    return list;
}
```

Deleting is also a bit treacherous. We need to find the cell to delete and then carefully splice it out of the list and free its memory. Again, we're going to carry two pointers down the list to find the cell to delete and the cell previous to it. If we don't find the cell at all, we bail. Once we have the cell and its previous neighbor, we can wire up everything to circumvent the cell being killed. Again, we have the special case of deleting the first cell in the list.

```
void deleteAddress(addressRec *& list, string name) {
    addressRec *cur;
    addressRec *prev = NULL;
    for (cur = list; cur != NULL; cur = cur->next) {
        int cmp = name.compare(cur->name);
        if (cmp == 0) break; // found it
        if (cmp < 0) return; // passed position, didn't find it
        prev = cur;
    }
    if (cur == NULL) return; // we never found it

    // now, cur points to the entry to delete, prev is one before
    if (prev != NULL)
        prev->next = cur->next;
    else
        list = cur->next; // recall that list is a reference ☺
    delete cur; // free all memory from this cell
}
```

You might note that find, inserting, deleting all start with a very similar loop, and a good instinct is to want to unify them into a helper function. This function could be given a list and a name and would find the position in the list that cell would be at. It could return the two surrounding cells by reference that you could use to finish off insert, delete, find, and so forth.

## The Josephus Problem: Take II

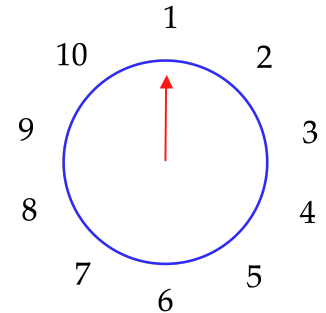
We solved this problem using a **Queue** a few weeks ago, but it's presented here again to illustrate how lightweight the algorithm is in comparison when you use an exposed, circularly linked list.

This problem is a variant of an ancient problem named for Flavius Josephus, a famous historian of the first century. Legend has it that Josephus wouldn't have lived to become

famous without his mathematical talents. During the Jewish-Roman war, he was among a band of 41 Jewish rebels exiled to a cave by the Romans. Preferring suicide to capture, the rebels decided to form a circle and, proceeding around it, to kill every other remaining person until no one was left. But Josephus, along with an unindicted co-conspirator, wanted none of this suicide nonsense, so he quickly calculated where he and his accomplice should stand in this circle so they were the last ones to be executed.

In our variation, we start with  $n$  people numbered **1** to  $n$  around a circle, and we eliminate every other person until only two survive. We're interested in the positions of the two survivors for any particular choice of  $n$ . For instance, presented to the right is the starting configuration for  $n$  equal to **10**. The elimination order is **2, 4, 6, 8, 10, 3, 7**, and then **1**, so **5** and **9** survive.

Using a circular, singly linked list as an auxiliary data structure, we can write a pair of routines to determine the position indices of the last two people to be executed, as a function of  $n$ .



First, we can write a function **createCircle** to create a sorted **singly linked, circular** list of integers from **1** to  $n$ , and then returns the address of the **node** storing the **1**. (We'll assume that  $n$  is always greater than or equal to 1.)

```

struct nodeRec {
    int position;
    nodeRec *next;
};

/**
 * Function: createCircle
 * -----
 * createCircle creates a sorted linked list of all the
 * numbers between 1 and the specified argument. The
 * list is circular and singly-linked, and the address of the
 * node containing the 1 is returned. createCircle only behaves for
 * positive values of n.
 */

nodeRec *createCircle(int n) {
    nodeRec *last;
    nodeRec *first = last = createPosition(n, NULL);
    for (int position = n - 1; position >= 1; position--)
        first = createPosition(position, first);

    last->next = first;
    return first;
}

nodeRec *createPosition(int position, nodeRec *nextPosition) {
    nodeRec *node = new nodeRec;
    node->position = position;
    node->next = nextPosition;
    return node;
}

```

Then, using this **createCircle** function above, we can write a function **josephusSurvivors** that takes the number of people in the circle and returns (by reference) the positions of the last two people to be executed. We'll assume that the number of people in the circle is always two or more. The best approach is to simulate the execution by looping around the linked list and removing every other node, keeping track of the two most recently deleted positions. When the last node is removed, the two most recently deleted positions correspond to the last two people executed. We take care not to orphan any memory—specifically, we don't leave any circularly linked list nodes lying around in the heap.

```

/**
 * Function: josephusSurvivors
 * -----
 * josephusSurvivors returns the positions of the last two people
 * executed when every other person in a circle of numPeople people
 * is executed. The number of people in the circle is passed in as
 * the first argument, and the positions of the two survivors are
 * returned by reference via the pointers passed in as arguments 2 and 3.
 * josephusSurvivors expects numPeople to be 2 or greater.
 */

void josephusSurvivors(int numPeople, int& lastKilled, int& nextToLastKilled) {
    lastKilled = nextToLastKilled = 0; // technically not necessary
    nodeRec *survivor = createCircle(numPeople);
    while (true) {
        nodeRec *victim = survivor->next;
        nextToLastKilled = lastKilled;
        lastKilled = victim->position;
        survivor->next = victim->next;
        delete victim;
        if (victim == survivor) return; // okay to examine address itself
        survivor = survivor->next;
    }
}

```