ESc 101: Fundamentals of Computing

Lecture 36-37

Apr 8 & 12, 2010

Truly Dynamic Space

- In many applications, one needs to maintain a collection of data elements from which insertions and deletions are made frequently.
- For example, maintaining a list of people entering the swimming pool in a particular slot.
- In such a situation, even during execution, one cannot predict the size of array required to store all the data.
- To maintain such dynamic collection, we use linked lists.

Linked Lists

- Each element, called cell, of a linked list stores some data, and also contains a pointer to the next element.
- The cells are not stored contiguously in the memory, and so cannot be accessed by indexing.
- This makes the access to lists slower than arrays.
- There are ways of making the access faster, but we will not go into that.

Defining a List

```
struct list {
    int element; // stores data
    struct list *next; // pointer to the next cell
}

typedef struct list *List; // defines a new type
```

The NULL Pointer

- To represent the case when a pointer does not point to a useful memory location, it is set to value `NULL`.
- It is pre-defined to be the memory address `0`.
- It is very useful to detect the end of the list: the pointer to the next element is `NULL`.

Inserting in a List

```
/* Inserts a number in the beginning of list L, and returns
 * a pointer to the inserted cell. L points to the first
 * cell of the list.
 */
List insert_list(int number, List L)
{
    List e;

    e = (List) malloc(sizeof(struct list)); // allocate space
    e->element = number; // store the element
    e->next = L; // points to the first cell
    return e; // new head of the list
}
```

Searching in a List

```
/* Searches for occurrence of a number in the list L.
 * Returns a pointer to that cell if it exists.
 * Returns NULL otherwise.
 */
List search_list(int number, List L)
{
    if (L == NULL) // empty list
        return L;

    if (L->element == number) // first cell matches!
        return L;

    // search remaining list
    return search_list(number, L->next);
}
```

Deleting Number from List

```
/* Deletes the number from the list L if it exists.
 * Returns the head of the list.
 */
List delete_list(int number, List L)
{
    List x, y;

    x = search_list(number, L); // look for the number

    if (x == NULL) // number does not exist
        return L;

    if (x == L) { // first cell to be deleted!
        L = L->next; // move the head
        free(x);
        return L;
    }
```

Deleting Number from List

```
    // find the cell before the one to be deleted
    for (y = L; y->next != x; y = y->next);

    y->next = x->next; // jump over x!
    free(x); // delete x
    return L;
}
```

Problems with `delete_list()`

- After searching for the cell to be deleted, the function has to make another pass to find the cell just before it.
  - Without the previous cell, the cell to be deleted cannot be properly removed from the list.
- The function does not return any information on whether the number is not found in the list.
  - This is because it has to return a pointer to the head of the list since it may change.

Fixing the Problems

- These problems can be fixed easily by maintaining a dummy cell at the head of the list that does not store any number.
- This cell is therefore never deleted, and so the pointer to the head never changes.
- In `search_list()`, we return a pointer not to the cell containing the number searched, but to the cell just before it.

Inserting in a List Again

```
/* Inserts a number in the beginning of list L. L points to
 * the first  * cell of the list - a dummy cell storing the
 * number of cells in the list.
 */
void insert_list(int number, List L)
{
    List e;

    e = (List) malloc(sizeof(struct list)); // allocate space
    e->element = number; // store the element
    e->next = L->next; // e points to the first cell
    L->next = e; // reassign the first cell
    (L->element)++; // increment the count
}
```

Searching in a List Again

```
/* Searches for occurrence of a number in the list L.
 * Returns a pointer to the cell just before the one
 * containing the number, if it exists.
 * Returns NULL otherwise.
 */
List search_list(int number, List L)
{
    if (L->next == NULL) // number does not exist
        return NULL;

    if ((L->next)->element == number) // found it!
        return L;

    // search remaining list
    return search_list(number, L->next);
}
```

Deleting Number from List Again

```
/* Deletes the number from the list L if it exists.
 * Returns -1 if number is not found.
 */
int delete_list(int number, List L)
{
    List x, y;

    x = search_list(number, L); // look for the number
        if (x == NULL) // number does not exist
        return -1;

    y = x->next; // cell to be deleted
    x->next = y->next; // jump over y!
    (L->element)--; // reduce the count
    free(y); // delete y
    return 1;
}
```

Using Lists

```
int main()
{
    List L; // empty list
    int number;
    char op;

    // create dummy cell
    L = (List) malloc(sizeof(struct list));
    L->next = NULL;
    L->element = 0;
```

Using Lists

```
    while (1) {
        scanf("%d %c", &number, &op);
        if (op == 'i')
            insert_list(number, L);
        else if (op == 'd')
            delete_list(number, L);
        else { // search
            if (search_list(number, L) == NULL)
            // does not exist
                printf("Number %d does not exist\n", number);
            else
                printf("Number %d exists\n", number);
        }
    }
}
```

Commands

`while:`	A loop construct.
`do-while:`	Another loop.
`continue:`	Stops the current iteration of a loop and starts the next iteration.
`switch:`	Alternative to `if-else` in some situations.
`goto:`	Jumps to a specified location of the program.

Advanced Features

`static variables:` Scope of these variables is different.

`bitwise operations:` Work on the individual bits of variables.

`unions:` Special types that can hold different sized and type of data.

`macros:` More commands beginning with #.

`low-level I/O:` Accessing files byte-by-byte.

`variable number of parameters:` Functions can have a variable number of parameters like `scanf()` and `printf()`.

`passing functions as arguments:` Pointers to functions can be passed as arguments to other functions!

From the next class, we start on Mathematica!