

# **Lecture 4: Elementary Data Structures**

**Steven Skiena**

Department of Computer Science  
State University of New York  
Stony Brook, NY 11794-4400

<http://www.cs.stonybrook.edu/~skiena>

## Problem of the Day

---

Find two functions  $f(n)$  and  $g(n)$  that satisfy the following relationship. If no such  $f$  and  $g$  exist, write “None”.

1.  $f(n) = o(g(n))$  and  $f(n) \neq \Theta(g(n))$
2.  $f(n) = \Theta(g(n))$  and  $f(n) = o(g(n))$
3.  $f(n) = \Theta(g(n))$  and  $f(n) \neq O(g(n))$
4.  $f(n) = \Omega(g(n))$  and  $f(n) \neq O(g(n))$

Recall that little oh  $o(n)$  means “grows strictly small than”.

# Elementary Data Structures

---

“Mankind’s progress is measured by the number of things we can do without thinking.”

Elementary data structures such as stacks, queues, lists, and heaps are the “off-the-shelf” components we build our algorithm from.

There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementation of these operations.

# Data Abstraction

---

That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.

- $Push(x,s)$  – Insert item  $x$  at the top of stack  $s$ .
- $Pop(s)$  – Return (and remove) the top item of stack  $s$ .

That there are different implementations of the same abstract operations enables us to optimize performance in different circumstances.

# Contiguous vs. Linked Data Structures

---

Data structures can be neatly classified as either *contiguous* or *linked* depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

# Arrays

---

An array is a structure of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

Advantages of contiguously-allocated arrays include:

- Constant-time access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.
- Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

# Dynamic Arrays

---

Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.

Compensating by allocating extremely large arrays can waste a lot of space.

With *dynamic arrays* we start with an array of size 1, and double its size from  $m$  to  $2m$  each time we run out of space.

How many times will we double for  $n$  elements? Only  $\lceil \log_2 n \rceil$ .

## How Much Total Work?

---

The apparent waste in this procedure involves the recopying of the old contents on each expansion.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements  $M$  is:

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Thus each of the  $n$  elements move an average of only twice, and the total work of managing the dynamic array is the same  $O(n)$  as a simple array.

Geometric series convergence is the free lunch of algorithm analysis.

# Pointers and Linked Structures

---

Pointers represent the address of a location in memory.

A cell-phone number can be thought of as a pointer to its owner as they move about the planet.

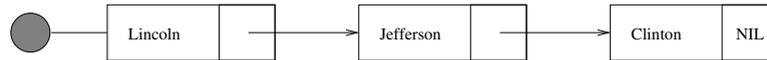
In C,  $*p$  denotes the item pointed to by  $p$ , and  $\&x$  denotes the address (i.e. pointer) of a particular variable  $x$ .

A special `NULL` pointer value is used to denote structure-terminating or unassigned pointers.

# Linked List Structures

---

```
typedef struct list {  
    item_type item;  
    struct list *next;  
} list;
```



# Searching a List

---

Searching in a linked list can be done iteratively or recursively.

```
list *search_list(list *l, item_type x)
{
    if (l == NULL) return(NULL);

    if (l->item == x)
        return(l);
    else
        return( search_list(l->next, x) );
}
```

## Insertion into a List

---

Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head.

```
void insert_list(list **l, item_type x)
{
    list *p;

    p = malloc( sizeof(list) );
    p->item = x;
    p->next = *l;
    *l = p;
}
```

Note the `**l`, since the head element of the list changes.

# Deleting from a List

---

```
delete_list(list **l, item_type x)
{
    list *p; (* item pointer *)
    list *last = NULL; (* predecessor pointer *)

    p = *l;
    while (p->item != x) { (* find item to delete *)
        last = p;
        p = p->next;
    }

    if (last == NULL) (* splice out of the list *)
        *l = p->next;
    else
        last->next = p->next;

    free(p); (* return memory used by the node *)
}
```

# Advantages of Linked Lists

---

The relative advantages of linked lists over static arrays include:

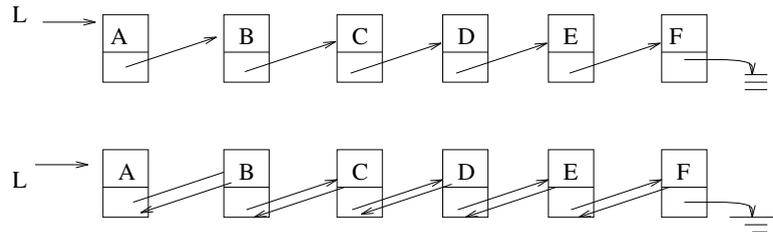
1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

# Singly or Doubly Linked Lists

---

We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.



Since the extra big-Oh costs of doubly-linked lists is zero, we will usually assume they are so maintained, although it might not always be necessary.

**Singly linked to doubly-linked list is as a conga line is to a can-can line.**

# Stacks and Queues

---

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

A *stack* supports last-in, first-out operations:

- $Push(x,s)$  – Insert item  $x$  at the top of stack  $s$ .
- $Pop(s)$  – Return (and remove) the top item of stack  $s$ .

A *queue* supports first-in, first-out operations:

- $Enqueue(x,q)$  – Insert item  $x$  at the back of queue  $q$ .
- $Dequeue(q)$  – Return (and remove) the front item from queue  $q$ .

# Stack/Queue Implementations

---

- Stacks are more easily represented as an array, with push/pop incrementing/decrementing a counter.
- Queues are more easily represented as a linked list, with enqueue/dequeue operating on opposite ends of the list.

All operations can be done in  $O(1)$  time for both structures, with both arrays and lists.

# Why Stacks and Queues?

---

Both are appropriate for a container class where order doesn't matter, but sometime it does matter.

Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

The entire difference between depth-first search (DFS) and breadth-first search (BFS) is whether a stack or a queue holds the vertices/items to be processed.

# Dictionary / Dynamic Set Operations

---

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- $Search(S, k)$  – A query that, given a set  $S$  and a key value  $k$ , returns a pointer  $x$  to an element in  $S$  such that  $key[x] = k$ , or nil if no such element belongs to  $S$ .
- $Insert(S, x)$  – A modifying operation that augments the set  $S$  with the element  $x$ .
- $Delete(S, x)$  – Given a pointer  $x$  to an element in the set  $S$ , remove  $x$  from  $S$ . Observe we are given a pointer to an element  $x$ , not a key value.

- $Min(S)$ ,  $Max(S)$  – Returns the element of the totally ordered set  $S$  which has the smallest (largest) key.
- **Logical**  $Predecessor(S,x)$ ,  $Successor(S,x)$  – Given an element  $x$  whose key is from a totally ordered set  $S$ , returns the next smallest (largest) element in  $S$ , or NIL if  $x$  is the maximum (minimum) element.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

**There is an inherent tradeoff between these operations. We will see that no single implementation will achieve the best time bound for all operations.**

## Array Based Sets: Unsorted Arrays

---

- $Search(S,k)$  – sequential search,  $O(n)$
- $Insert(S,x)$  – place in first empty spot,  $O(1)$
- $Delete(S,x)$  – copy  $n$ th item to the  $x$ th spot,  $O(1)$
- $Min(S,x)$ ,  $Max(S,x)$  – sequential search,  $O(n)$
- $Successor(S,x)$ ,  $Predecessor(S,x)$  – sequential search,  $O(n)$

## Array Based Sets: Sorted Arrays

---

- $Search(S,k)$  – binary search,  $O(\lg n)$
- $Insert(S,x)$  – search, then move to make space,  $O(n)$
- $Delete(S,x)$  – move to fill up the hole,  $O(n)$
- $Min(S,x)$ ,  $Max(S,x)$  – first or last element,  $O(1)$
- $Successor(S,x)$ ,  $Predecessor(S,x)$  – Add or subtract 1 from pointer,  $O(1)$