Lecture 4:
Elementary Data Structures

Steven Skiena

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

http://www.cs.sunysb.edu/∼skiena

Problem of the Day

True or False?

1. $2n^2 + 1 = O(n^2)$
2. $\sqrt{n} = O(\log n)$
3. $\log n = O(\sqrt{n})$
4. $n^2(1 + \sqrt{n}) = O(n^2 \log n)$
5. $3n^2 + \sqrt{n} = O(n^2)$
6. $\sqrt{n} \log n = O(n)$
7. $\log n = O(n^{-1/2})$

The Base is not Asymptotically Important

Recall the definition, $c^{\log_c x} = x$ and that

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus $\log_2 n = (1/\log_{100} 2) \times \log_{100} n$. Since $1/\log_{100} 2 = 6.643$ is just a constant, it does not matter in the Big Oh.

Federal Sentencing Guidelines

2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.
(a) Base offense Level: 6
(b) Specific offense Characteristics

(1) If the loss exceeded $2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

Make the Crime Worth the Time

The increase in punishment level grows *logarithmically* in the amount of money stolen.

Thus it pays to commit one big crime rather than many small crimes totalling the same amount.

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.

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

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 given by

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

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.

Pointers and Linked Structures

Pointers represent the address of a location in memory.

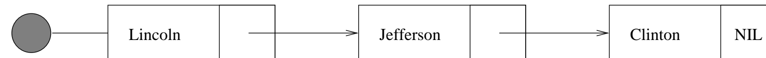
A cell-phone number can be thought of as a pointer to its owner as they move about the planet.

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.

Stacks and Queues

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

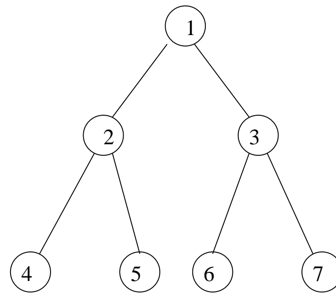
A *stack* supports last-in, first-out operations: push and pop.

A *queue* supports first-in, first-out operations: enqueue and dequeue.

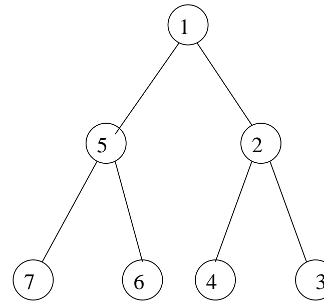
Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

Impact on Tree Traversal

Both can be used to store nodes to visit in a tree, but the order of traversal is completely different.



Queue



Stack

Which order is friendlier for WWW crawler robots?

Dictonary / 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.
- *Next(S,x), Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) 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.

Array Based Sets: Unsorted Arrays

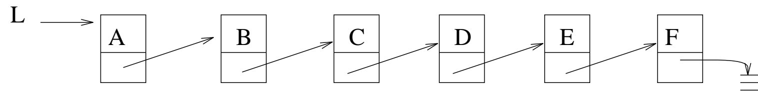
- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot, $O(1)$
- Delete(S,x) - copy nth item to the xth spot, $O(1)$
- Min(S,x), Max(S,x) - sequential search, $O(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $O(n)$

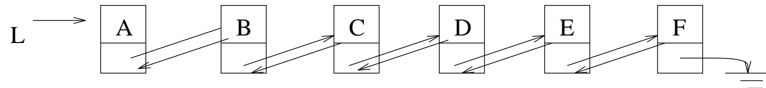
Array Based Sets: Sorted Arrays

- 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, $O(1)$

Pointer Based Implementation

We can maintain a dictionary in either a singly or doubly linked list.





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-linkly lists is zero, we will usually assume they are, although it might not be necessary.

Singly linked to doubly-linked list is as a Conga line is to a Can-Can line.