CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC03
ELEMENTARY DATA STRUCTURE (65-77)

Lecture slide courtesy of Prof. Steven 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. \(\log n = O(n^{-1/2})\)
Elementary data structures such as stacks, queues, lists, and heaps are the “off-the-shelf” components we build our algorithm from.

Changing the data structure does not change the correctness of the program. However, it can change the total performance time.

There are two aspects to any data structure:

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

containers, dictionaries, and priority queues
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
    - Ex> arrays, matrices, heaps, and hash tables.
  - Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers*
    - Ex> 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.
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 $\text{ceil}(\log_2 n)$. 

**Dynamic Arrays**
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:

$$M = \sum_{i=1}^{\log n} i \cdot \frac{n}{2^i} = n \sum_{i=1}^{\log n} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

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

1. one or more data fields
2. a pointer field to at least one other node
3. pointer to the head of the structure
Searching in a linked list can be done iteratively or recursively.

Recursive implementation:

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

```c
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 - RECURSIVE

```c
list *predecessor_list(list *l, item_type x) {
    if ((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought on null list.\n");
        return(NULL);
    }

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

delete_list(list **l, item_type x) {
    list *p; /* item pointer */
    list *pred; /* predecessor pointer */
    list *search_list(), *predecessor_list();

    p = search_list(*l,x);
    if (p != NULL) {
        pred = predecessor_list(*l,x);
        if (pred == NULL) /* splice out out list */
            *l = p->next;
        else
            pred->next = p->next;

        free(p); /* free memory used by 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.
Sometimes, the order in which we retrieve data is *independent of its content*, being only a function of when it arrived. (DS called *Containers*)

A **stack** supports last-in, first-out (LIFO) 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 (FIFO) operations:
- `Enqueue(x,q)`: Insert item x at the back of queue q.
- `Dequeue(q)`: Return (and remove) the front item from queue q.

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

Stacks and queues can be effectively implemented using either arrays or linked lists.
Impact on Tree Traversal

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

- Which order is friendlier for WWW crawler robots?
The dictionary data type permits access to data items by content.

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 \(\text{key}[x] = k\), 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**\((S,\ast x)\) – 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.
May also have following functions:

- \( \text{Min}(S), \text{Max}(S) \) – Returns the element of the totally ordered set \( S \) which has the smallest (largest) key.
- \( \text{Next}(S,k), \text{Previous}(S,) \) – Retrieve the item from \( D \) whose key is immediately before (or after) \( k \) in sorted order.

These enable us to iterate through the elements of the data structure. 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), Max(S) - sequential search, O(n)
- Successor(S,k), Predecessor(S,k) - 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), **Max**(S) - first or last element, $O(1)$
- **Successor**(S,k), **Predecessor**(S,k) - Add or subtract 1 from pointer, $O(1)$
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-linked 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.
Problem: What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list.
- A doubly-linked unsorted list.
- A singly-linked sorted list.
- A doubly-linked sorted list.
<table>
<thead>
<tr>
<th>Dictionary operation</th>
<th>Singly unsorted</th>
<th>Double unsorted</th>
<th>Singly sorted</th>
<th>Doubly sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search($L, k$)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insert($L, x$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Delete($L, x$)</td>
<td>$O(n)^*$</td>
<td>$O(1)$</td>
<td>$O(n)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Successor($L, x$)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Predecessor($L, x$)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Minimum($L$)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Maximum($L$)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>