Introduction to data structures

William Hendrix

Lecture 9

Outline

- Overview
- Stacks and Queues
- Dictionaries
  - Arrays vs. linked-lists
  - Sorted vs. unsorted
  - Binary search trees
  - Hash tables
  - Bit vectors
- Heaps
- Union-find
- Prefix and suffix trees

Why are we studying data structures?

- Data structures are a fundamental component of algorithms
- Different data structures support different operations
- Some have multiple implementations
- Strongly affect complexity
  - **Example:** SelectionSort

    Input: *data*: an array of integers to sort
    Input: n: the number of values in data
    Output: permutation of data such that $data[1] \leq \ldots \leq data[n]$

    Algorithm: SelectionSort

    for $i = 1$ to n **do**
        Let m be the location of the min value in the array $data[i..n]$;
        Swap $data[i]$ and $data[m]$;
    end
    return *data*;

  - Heap: data structure that takes $O(n)$ time to build and $O(\lg n)$ time to delete the min
  - Transforms $O(n^2)$ SelectionSort into $O(n \lg n)$ HeapSort

Data structures in memory

- Contiguous
  - Allocate single chunk of memory
  - Retrieve elements by locating data's index in chunk
  - Very fast if elements are accessed consecutively (caching)
  - No memory wasted on storing pointers
  - Following *k* links takes *O(k)* time vs. *O(1)* for pointer arithmetic
- Link-based
  - Data are stored in small "islands" connected via pointers
  - Retrieve elements by starting at the head/tail/root and traversing links
  - Supports data with irregular structure (graphs, trees)
  - Easier for memory manager to allocate
  - Easy to modify structure by changing links (vs. copying data)
- Hybrid

Stacks and queues

- Abstract data structures
  - Define a set of operations
  - Generally implemented as arrays
- **Stacks**
  - Support *push()* and *pop()* operations
  - Last-In, First-Out (LIFO) order
  - Clears out "new" work quickly
  - DFS generally requires less space than BFS

- **Queues**
  - Support *enqueue()* and *dequeue()* operations
  - First-In, First-Out (FIFO) order
  - More "fair": delay between *enqueue()* and *dequeue()* balanced
- **Deques** ("decks")
  - Support all 4 operations

Operation analysis

- **push(x)**
  - Depends on *capacity*
    - Allocated memory
  - If capacity > size, just add
  - Otherwise, enlarge first
    - Copy all elements over
    - *O(n) time*

```
Algorithm: Push(x)
if n ≥ c then
|   Enlarge();
end
n = n+1;
stack[n] = x;
```

```
Algorithm: Enlarge()
c = ???;
newstack = allocate(c);
for i = 1 to n do
|   newstack[i] = stack[i]
end
free stack;
stack = newstack;
```

Array enlargement policy

- **Bad policy:** increment by 1
  - Cost: O(n) every time
  - n pushes: $O(1+2+\ldots+n)) = O(n^2)$
  - *Average cost per push:* $O(n)$
- **Better policy:** increment by k (k=10, 100, etc.)
  - Cost: O(n) every k^{th} time
  - n pushes: $O(1+(1+k)+\ldots+(1+k\lfloor\frac{n}{k}\rfloor)) = O\left(\frac{n^2}{k}\right)$
  - *Average cost per push:* $O(\frac{n}{k})$
  - *Caution*: space trade-off
    - Increment by 1B: small stacks will be mostly empty space
- **Best policy:** double array size
  - Cost: O(n) after powers of two
  - n pushes: $O(1+2+4+\ldots+2^{\lfloor \lg n \rfloor}) = O(n)$
  - *Average cost per push:* $O(1)$
  - Array will be at least half-full if not deleting

Amortized analysis

Operation analysis

- **push(x)**
  - Depends on *capacity*
    - Allocated memory
  - If capacity > size, just add
  - Otherwise, enlarge first
  - *O(1) time, amortized*
- **pop()**
  - Remove last element
  - *O(1) time*
- **enqueue(x)**
  - Similar to push()
  - *O(1) time, amortized*
- **dequeue(x)**
  - Advance "head" pointer
  - *O(1) time*

```
Algorithm: Push(x)
if n ≥ c then
    Enlarge();
end
n = n+1;
stack[n] = x;
```

```
Algorithm: Enlarge()
c = 2c;
newstack = allocate(c);
for i = 1 to n do
    newstack[i] = stack[i]
end
free stack;
stack = newstack;
```

```
Algorithm: Pop()
if n = 1 then
    error "Stack is empty";
end
n = n - 1;
return data[n];
```

Dictionary

- Abstract data structure for storing and retrieving values

- **Primary operations**
  - *Search(x)*: returns the location of *x* in the dictionary, or NIL if not contained
  - *Insert(x)*: adds *x* to the dictionary
  - *Delete(x)*: removes *x* from the dictionary

- **Additional operations**
  - *Max(), Min()*: return the location of the largest/smallest element
  - *Successor(x), Predecessor(x)*: return the next largest/smallest element than *x*

Array-based dictionary operations

- *Unsorted* array

- **Search(x)**
  - Linear scan
  - *O(n) time*

```
Algorithm: Search(x)
for i = 1 to n do
    if dict[i] = x then
        return &dict[i];
end
return NIL;
```

- **Delete(x)**
  - Swap "victim" with last node
  - *O(1) time*

```
Algorithm: Delete(x)
*x = dict[n];
n = n - 1;
```

Array-based insertion

- *Unsorted* array
- **Insert(x)**
  - Depends on capacity (*c*)
  - If capacity is available:
    - Add element to next position
    - *O(1) time*
  - If space is not available:
    - Allocate larger array
    - Copy elements from old array
    - Deallocate old array
    - Insert as normal
    - *O(1) time, amortized*

```
Algorithm: Insert(x)
if n ≥ c then
|  Enlarge();
end
n = n+1;
dict[n] = x;
```

```
Algorithm: Enlarge()
c = 2c;
newdict = allocate(c);
for i = 1 to n do
|  newdict[i] = dict[i]
end
free dict;
dict = newdict;
```

Sorted array operations

- **Search(x)**
  - Binary search
  - *O(lg n) time*

```
Algorithm: Search(x)
lo = 1;
hi = n;
while lo < hi do
    mid = floor((lo + hi)/2);
    if data[mid] = x then
        return &data[mid];
    else if data[mid] > x then
        lo = mid+1;
    else
        hi = mid-1;
    end
end
if lo ≤ n and data[lo] = x then
    return &data[lo];
else
    return NIL;
end
```

Sorted array operations

- **Delete(x)**
  - Shift elements left
  - *O(n) time*

```
Algorithm: Delete(x)
for i = x-data to n-1 do
    data[i] = data[i+1];
end
n = n-1;
```

- **Insert(x)**
  - Enlarge (if needed), then shift right
  - *O(n) time*

```
Algorithm: Insert(x)
if n ≥ c then
    Enlarge();
end
Use binary search to find i such
that data[i] ≤ x ≤ data[i+1];
for j = n to i+1 step -1 do
    data[j+1] = data[j];
end
data[i+1] = x;
n = n+1;
```

- ***Conversion from unsorted array***
  - Sort
  - *O(n lg n) time*

Summary: array-based dictionaries

Operation	Unsorted array	Sorted array
Search(x)	*O(n)*	*O(lg n)*
Delete(x)	*O(1)*	*O(n)*
Insert(x)	*O(1), amortized*	*O(n)*
Build	n/a	*O(n lg n)*
Min()	*O(n)*	*O(1)*
Max()	*O(n)*	*O(1)*
Predecessor(x)	*O(n)*	*O(1)*
Successor(x)	*O(n)*	*O(1)*

List-based dictionaries

- Unsorted linked list
  - Largely similar to unsorted array
  - **Search(x)**
    - Linear scan
    - *O(n) time*
  - **Insert(x)**
    - Append element
    - *O(1) time,* with tail pointer
  - **Delete(x)**
    - Depends!
    - Singly-linked list: linear scan
      - *O(n) time*

```
Algorithm: Delete-SLL(x)
if x = head then
    head = x.next;
    if x = tail then
        tail = NIL;
    free x;
else
    curr = head;
    while curr ≠ NIL and curr.next
    ≠ x do
        curr = curr.next;
    end
    if curr ≠ NIL then
        curr.next = x.next;
        if x = tail then
            tail = curr;
        free x;
    end
end
```

List-based dictionaries

- Unsorted linked list
  - Largely similar to unsorted array
  - **Search(x)**
    - Linear scan
    - *O(n) time*
  - **Insert(x)**
    - Append element
    - *O(1) time,* with tail pointer
  - **Delete(x)**
    - Depends!
    - Singly-linked list: linear scan
      - *O(n) time*
    - Doubly-linked list: pointer ops
      - *O(1) time*

```
Algorithm: Delete-DLL(x)
if x = head then
    head = x.next;
    if head ≠ tail then
        x.next.prev = NIL;
    free x;
else if x = tail then
    tail = x.prev;
    x.prev.next = NIL;
    free x;
else
    x.prev.next = x.next;
    x.next.prev = x.prev;
    free x;
end
```

Sorted linked lists

- **Search(x)**
  - No link to midpoint, so no binary search!
  - Linear scan
  - *O(n) time*
- **Insert(x)**
  - Linear scan
  - *O(n) time*
- **Delete(x)**
  - Same as unsorted
  - Singly-linked list: *O(n) time*
  - Doubly-linked list: *O(1) time*

- **Building a sorted list**
  - Possible in *O(n lg n)* time

Summary: link-based dictionaries

Operation	Unsorted SLL	Unsorted DLL	Sorted SLL	Sorted DLL
Search(x)	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Delete(x)	*O(n)*	*O(1)*	*O(n)*	*O(1)*
Insert(x)	*O(1)*	*O(1)*	*O(n)*	*O(n)*
Build	n/a	n/a	*O(n lg n)*	*O(n lg n)*
Min()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Max()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Predecessor(x)	*O(n)*	*O(n)*	*O(n)*	*O(1)*
Successor(x)	*O(n)*	*O(n)*	*O(1)*	*O(1)*

- **Note:** DLL time is strictly better, asymptotically
  - Trade-off: more space, more pointer manipulation

Coming up

- More data structures!

- **Homework 6** (posted tonight) will be due Thursday
- **Project 1** will be posted this weekend
  - Search algorithms, Big-Oh analysis
- **Exam 1** will be returned next week

- **Recommended readings (today):** Sections 3.1-3.4
- **Recommended readings (Tuesday):** Sections 3.5-3.9
- **Practice problems:** 1-2 problems from "Stacks, Queues, and Lists"