(Algorithms and) Data Structures

Gabriel Istrate

March 1, 2024

First of all ...

Organizational

- Course objectives: present main data structures, improve alg. design.
- Course webpage: *moodle.unibuc.ro*. Self-register.
- Class handouts: *will be posted on webpage.*
- Lab: Bianca Mogoş + others.
- Each seminarist: **responsible for their own rules concerning seminar/lab grade.**

Organizational (II)

- Grading: exam, (lab) homework/programming assignments.
- Exact percentages: still undefined, probably 66%-33 %.
- Course attendance: strongly recommended, not strictly enforced.

However

being a student is a full time job. Would your boss accept “couldn’t do X because I had classes ?”.

- Expect you: Work hard.
- Exam: **no point in memorizing courses, there will be no ”theory” part as such.** Rather, I want to see that you understood material.

Important

- Academic honesty
- OK/encouraged: speak up in class. Two-way, rather than one-way communication. Request: *be concise, to the point, respect time spent together in class.*
- Disclaimer: I can make mistakes/be wrong. Let me know (in person, email) how I can improve things.
- **Also:** Please be in good standing with the university. I am **not** competent to answer questions of this nature (I will give you a grade if you appear in the catalog; it's your responsibility to be there)

Organizational (III)

- less programming, more math.
- Reason: need to know pointers, you're learning them this semester.
- However: *mostly C, some basic features of C++, STL.*
- Please: brush up on/read **basic features** of C/C++, I will review them in the Lab sections.
- video lectures from previous years (linked on classroom).

Textbook: combination of texts.

- **Cormen, Leiserson, Rivest, Stein**. (3rd ed.) First edition translated in Romanian as well. Theoretical, but very good book.
- (secondary) Adam Drozdek *"Data structures and algorithms in C++", third edition or newer.* Good for programming.

Expect you to read from textbook, not only from slides !

Where are we ?

Several methods for designing algorithms

- Divide and conquer.
- Greedy.
- Dynamic programming.
- (when everything fails) Backtracking.

Data structures

Paradigm for developing efficient algorithms based on abstraction: Abstract operations needed by the program, change implementation so that frequent operations run fast.

Example: Selection sort vs. HEAPSORT

Selection SORT

- Elements to sort in a vector.
- Find maximum element.
- Swap it with the last element.
- Proceed recursively.

Complexity

- Finding maximum in a vector: $\Theta(n)$.
- Complexity analysis: $T(n) = T(n-1) + \theta(n)$.
- Conclusion: $\Theta(n^2)$

Example: Selection sort vs. HEAPSORT

Selection SORT

- Bottleneck in Selection Sort: Find maximum element
- If we could improve finding max

HEAPSORT

- Algorithm: Same idea.
- Bottleneck: FindMax $\Theta(n)$. Replace it with $O(1)$ operation (via heaps).
- Complexity $W(n) = \theta(\log n)$ (need also to update heap via HEAPIFY)
- New algorithm: complexity $\Theta(n \log(n))$.

Why data structures ?

- Data structures: algorithm development via primitive operations.

Modularity in algorithm design

You don't build a house from scratch (bricks, frames, drywalls). Same with algorithms/code.

- Easier to solve problem/test solution only once.
- Correctness: easier to check. easier to update.

Why data structures ? Performance.

- You google something. **Don't want to wait 100 seconds !** Search: fast.
- You play a game. Game engine must quickly retrieve/update objects you see in front of you when you move your viewport.
- Operations: often abstracted from requirements.

Most frequent operations should be fast.

How do you measure performance ?

$O(n \log n)$, $\Theta(\log n)$ ...

Example (operations from requirements)

$$\underbrace{1\ \ 2\ \ 3}\ \ \underbrace{7\ \ 5\ \ 6\ \ 4}\ \ \underbrace{8\ \ 9\ \ 10}\ \ \underbrace{14\ \ 13\ \ 12\ \ 11}\ldots$$

- TCP: basis for much of Internet traffic.
- Data requirement: We need to *buffer* a packet that is out-of order.
- We need to *pop* elements that become in-order.
- We need to *test emptiness of buffer*.
- We need to produce *first missing element* (ACK).
- Operation performance $O(1)$?.

Concepts

- A data structure is a way to organize and store information
  - to facilitate access, or for other purposes
- A data structure has an interface consisting of procedures for adding, deleting, accessing, reorganizing, etc.
- A data structure stores data and possibly meta-data
  - e.g., a *heap* needs an array A to store the keys, plus a variable $A.\mathit{heap\text{-}size}$ to remember how many elements are in the heap

What are data structures more concretely ?

- data ...
- E.g. complex numbers: two floats.
- ... together with operations one can perform on the data ...
  Example: integer + (addition), - (subtraction), $\cdot$ (multiplication).
- ... and performance guarantees.

Note !

How to precisely implement operations is **not** a part of data structure specification. Concepts, not code.

Data types

- All DS that share a common structure and expose the same set of operations.
- Predefined data types: array, structures, files.
- Scalar data type: ordering relation exists among elements.
- More complicated: dynamic DS. Lists, circular lists, trees, hash tables, graphs.

C++: Standard template library (STL):

library of container classes, algorithms, and iterators; provides many of the basic algorithms and data structures of computer science

Example: Array data type/Vector

- Ensures random access to its elements.
- **Complexity** $O(1)$.
- Composed of objects of the same type.

Implementations

- `int myarray[10];` One dimensional arrays.
- Multidimensional arrays.
  `type name[`lim_1`]...[`lim_n`];`
- implementation in C++/STL: *vector*.

Example using vector class

```
#include<vector>
using namespace std;

int main(){

static const int SIZE = 10000;
vector<int> arr( SIZE );
arr.append(125);
....
}
```

Vectors the C++/data structures way

- Vector: black-box.
- Random access: $arr[i]$ should take $\Theta(1)$ time.
- Black box (class implementation) may implement some other operations, e.g. append.

Main point

You didn't implement vector yourself. All you care is what operations can you execute, and how complex they are.

This course

Define, implement various "data structures", and use them to get better algorithms.

Some minimal C/C++ recap

You have an entire course for more.

Pointers in C(++)

Variables that hold *addresses* of other variables.

```
int i=15,j, *p,*q;
```

Dynamic memory allocation: `p= new int;`
Assignment: `*p=20;`
Deallocation: `delete p;`
Dangling reference: upon deallocation should assign $p = 0$;

Pointers and arrays

```
 int a[5],*p;


for(sum=a[0],i=1;i<5;i++)
 sum += a[i];
or
for (sum=*a,i=1;i<5;i++)
 sum += *(a+i);
or
for(sum=*a,p=a+1;p<a+5;p++)
 sum += *p;

p = new int[n];
delete [] p;
```

Pointers and reference variables

`int` $n = 5, *p = \&n, \&r = n;$.
r is a reference variable. Must be initialized in definition as reference to a particular variable.

reference: different name for/constant pointer to variable.

```
cout << n << ' '<< *p<<' '<< r<< endl;
```

5 5 5

```
n= 7 (*p = 7, r = 7)
cout << n << ' '<< *p<<' '<< r<< endl;
```

7 7 7

cout: C++ way to print. BEST WAY TO PASS PARAMETER: const reference variables;

C++: classes, objects, member functions, oh my !

- in C++: classes - user-defined data types.
- objects: instantiations of classes.
- objects have behavior, member functions.

Example

- Assume **dog** is a C++ class.
- Assume **Buddy** is an "object" of type dog.
- Dogs behavior: bark, member function with no parameters.
- To make Buddy bark: Buddy.bark() call member function bark that belongs to Buddy.
- C++: only so-called public member functions can be called from outside the class code.

So let's start ...

Today:

- Stacks
- Queues
- Dequeues

Stacks

- A **Stack** is a sequential organization of items in which the last element inserted is the first element removed. They are often referred to as LIFO, which stands for "last in first out."
- **Examples:** letter basket, stack of trays, stack of plates.
- The *only* element of a stack that may be accessed is the one that was most recently inserted.
- There are only two basic operations on stacks, the *push* (insert), and the *pop* (read and delete).

Stacks: Push and Pop

- The operation **push(x)** places the item x onto the top of the stack.
- The operation **pop()** removes the top item from the stack, and returns that item.
- We need some way of detecting an empty stack (This is an *underfull* stack).
  - In some cases, we can have **pop()** return some value that couldn't possibly be on the stack.
  - **Example:** If the items on the stack are positive integers, we can return "-1" in case of underflow.
  - In other cases, we may be better off simply keeping track of the size of the stack.
- In some cases, we will also have to worry about filling the stack (called *overflow*). One way to do this is to have **push(x)** return "1" if it is successful, and "0" if it fails.

An Example Stack Operations

Assume we have a stack of size 3 which holds integers between -100 and 100. Here is a series of operations, and the results.

Operation	Stack Contents	Return
create	()	
push(55)	(55)	1
push(-7)	(-7,55)	1
push(16)	(16,-7,55)	1
pop	(-7,55)	16
push(-8)	(-8,-7,55)	1
push(23)	(-8,-7,55)	0
pop	(-7,55)	-8
pop	(55)	-7
pop	()	55
pop	()	101

Array-based Stack Implementation

- S is an array that holds the elements of the stack
- $S.top$ is the current position of the top element of S

STACK-EMPTY(S)

if $S.top == 0$
    return TRUE
else return FALSE

PUSH(S,X)

$S.top = S.top + 1$
$S[S.top] = x$

POP(S)

if STACK-EMPTY(S)
    error “underflow”
else $S.top = S.top - 1$
    return $S[S.top + 1]$

Questions for you

1. What is the result of running operations PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8), POP(S) on an empty stack of size 6 ?
2. Show how to implement a queue with two stacks. Analyze the running time of the queue operations.
3. Explain how two implement two stacks in one array $A[1 \ldots n]$ such that neither stack overflows unless the total number of elements in both stacks is n. PUSH and POP should run in $O(1)$ time.

Example Application of Stacks: Balanced Parant

- Stacks can be used to check a program for balanced symbols (such as $\{\}$,(),[ ]).
- **Example:** $\{()\}$ is legal, as is $\{()(\{\})\}$, whereas $\{((\}$ and $\{(\})$ are not (so simply counting symbols does not work).
- If the symbols are balanced correctly, then when a closing symbol is seen, it should match the "most recently seen" unclosed opening symbol. Therefore, a stack will be appropriate.

Balanced symbols

The following algorithm will do the trick:

- While there is still input:

  s = next symbol
  if (s is an opening symbol) push(s)
  else //s is a closing symbol

    if (Stack.Empty) report an error
    else r = pop()
    if(! Match(s,r)) report an error

- If (! Stack.Empty) report an error

Examples

1. Input: { ( ) }
   - Read {, so push {
   - Read (, so push (. Stack has { (
   - Read), so pop. popped item is (which matches). Stack has now {.
   - Read }, so pop; popped item is { which matches }.
   - End of file; stack is empty, so the string is valid.
2. Input: { ( ) ({) } } (This will fail.)
2. Input: { ({ }){ } ( ) } (This will succeed.)
3. Input: { ( ) }) (This will fail.)

Operator Precedence Parsing

- We can use the stack class we just defined to parse and evaluate mathematical expressions like:

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

- First, we transform it to postfix notation:

$$5\ 9\ 8 + 4\ 6 * * 7 + *$$

- Usual form for arithmetic expressions: infix. term1 op term2.
- Postfix notation: term1 term2 op.
- How to convert infix to postfix: later !

Evaluating Postfix expressions

Then, the following C++ routine uses a stack to perform this evaluation:

```
char c;
Stack acc(50);
int x;
while (cin.get(c))
{
x = 0;
while (c == ' ' ) cin.get(c);
if (c == '+') x = acc.pop() + acc.pop();
if (c == '*') x = acc.pop() * acc.pop();
while (c≥ '0' && c ≤ '9')
x = 10*x + (c-'0'); cin.get(c);
acc.push(x);
}
cout << acc.pop() << ”;
```

Explanation of code

- We read one character at a time in c.
- In x we compute the value of the currently evaluated expression.
- After computing it we push the value on the stack - we will need it later.
- When reading an op we take the last two value off the stack and apply the op on them and assign this to x.
- When reading a digit we update value of x by making the last read digit the least significant one.

Stack Applications

- Recursion removal can be done with stacks.
- Reversing things is easily done with stacks.
- Procedure call and procedure return is similar to matching symbols:
  - When a procedure returns, it returns to the most recently active procedure.
  - When a procedure call is made, save current state on the stack. On return, restore the state by popping the stack.

Queue

- The ubiquitous "first-in first-out" container (FIFO)
- *Interface*
  - ENQUEUE(Q, x) adds element x at the back of queue Q
  - DEQUEUE(Q) extracts the element at the head of queue Q
- *Implementation*
  - Q is an array of fixed length $Q.length$
    - i.e., Q holds at most $Q.length$ elements
    - enqueueing more than Q elements causes an "overflow" error
  - $Q.head$ is the position of the "head" of the queue
  - $Q.tail$ is the first empty position at the tail of the queue

Enqueue

```
ENQUEUE(Q,x)
  if Q.queue-full
      error "overflow"
  else Q[Q.tail] = x
      if Q.tail < Q.length
          Q.tail = Q.tail + 1
      else Q.tail = 1
      if Q.tail == Q.head
          Q.queue-full = TRUE
      Q.queue-empty = FALSE
```

Enqueue

ENQUEUE(Q,x)

```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = TRUE
    Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

```
ENQUEUE(Q,x)
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = TRUE
    Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

ENQUEUE(Q,x)
```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = TRUE
    Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

```
ENQUEUE(Q,x)
    if Q.queue-full
        error "overflow"
    else Q[Q.tail] = x
        if Q.tail < Q.length
            Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
            Q.queue-full = TRUE
        Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

```
ENQUEUE(Q,x)
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = TRUE
    Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

```
ENQUEUE(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = TRUE
        Q.queue-empty = FALSE
```

Q.head
Q.tail

Enqueue

ENQUEUE(Q,X)

```
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = TRUE
    Q.queue-empty = FALSE
```

Q.head
Q.tail

Deque

```
DEQUEUE(Q)
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = TRUE
    Q.queue-full = FALSE
    return x
```

Dequeue

DEQUEUE(Q)

if $Q.\mathit{queue\text{-}empty}$
    error "underflow"
else $x = Q[Q.\mathit{head}]$
    if $Q.\mathit{head} < Q.\mathit{length}$
        $Q.\mathit{head} = Q.\mathit{head} + 1$
    else $Q.\mathit{head} = 1$
    if $Q.\mathit{tail} == Q.\mathit{head}$
        $Q.\mathit{queue\text{-}empty} = \text{TRUE}$
    $Q.\mathit{queue\text{-}full} = \text{FALSE}$
    return x

$Q.\mathit{head}$
$Q.\mathit{tail}$

Dequeue

```
DEQUEUE(Q)
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = TRUE
    Q.queue-full = FALSE
    return x
```

Q.head
Q.tail

Dequeue

```
DEQUEUE(Q)  if Q.queue-empty
                error "underflow"
            else x = Q[Q.head]
                if Q.head < Q.length
                    Q.head = Q.head + 1
                else Q.head = 1
                if Q.tail == Q.head
                    Q.queue-empty = TRUE
                Q.queue-full = FALSE
                return x
```

Q.head
Q.tail

Dequeue

DEQUEUE(Q)

if *Q.queue-empty*
    error "underflow"
else $x = Q[Q.head]$
    if $Q.head < Q.length$
        $Q.head = Q.head + 1$
    else $Q.head = 1$
    if $Q.tail == Q.head$
        $Q.queue\text{-}empty =$ TRUE
    $Q.queue\text{-}full =$ FALSE
    return x

$Q.head$
$Q.tail$

Applications of Queues

- Scheduling (disk, CPU)
- Used by operating systems to handle congestion.
- Algorithms (we'll see): breadth-first search.

Stacks,Queues: Scorecard

Algorithm	*Complexity*
STACK-EMPTY	$O(1)$ ✓
PUSH	$O(1)$ ✓
POP	$O(1)$ ✓
ENQUEUE	$O(1)$ ✓
DEQUEUE	$O(1)$ ✓
RESTRICTIONS:	LIFO/FIFO orders only. $\times$

Deques

- Like queues but can enqueue/dequeue at both ends.
- Can modify the code for queues, add two more procedure.
- do it !
- Complexity scorecard: similar to queues.

Dynamic sets

Major problem this semester:

Represent a set S whose elements may vary through time. May want to perform some of:

- INSERT(S,x)
- DELETE(S,x)
- SEARCH(S,x). Result YES/NO. Better: handle for x, if found.
- MIN(S)
- MAX(S)
- SUCC(S,x), PRED(S,x)

Example: stacks/queues

- Stacks: dynamic sets with LIFO order.
- Queues: dynamic sets with FIFO order.

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - $\textsc{Insert}(D, k)$ adds a key k to the dictionary D
  - $\textsc{Delete}(D, k)$ removes key k from D
  - $\textsc{Search}(D, k)$ tells whether D contains a key k

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - $\textsc{Insert}(D, k)$ adds a key k to the dictionary D
  - $\textsc{Delete}(D, k)$ removes key k from D
  - $\textsc{Search}(D, k)$ tells whether D contains a key k
- *Implementation*
  - many (concrete) data structures

Dictionary

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - INSERT(D, k) adds a key k to the dictionary D
  - DELETE(D, k) removes key k from D
  - SEARCH(D, k) tells whether D contains a key k
- *Implementation*
  - many (concrete) data structures
  - we'll see: hash tables

Direct-Address Table

- A *direct-address table* implements a dictionary

Direct-Address Table

- A *direct-address table* implements a dictionary
- The *universe* of keys is $U = \{1, 2, \ldots, M\}$

Direct-Address Table

- A *direct-address table* implements a dictionary

- The *universe* of keys is $U = \{1, 2, \ldots, M\}$

- *Implementation*
  - an array T of size M
  - each key has its own position in T

Direct-Address Table

- A *direct-address table* implements a dictionary
- The *universe* of keys is $U = \{1, 2, \ldots, M\}$
- *Implementation*
  - an array T of size M
  - each key has its own position in T

$$\text{Direct-Address-Insert}(T, k)1 \quad T[k] = \text{TRUE}$$

$$\text{Direct-Address-Delete}(T, k)1 \quad T[k] = \text{FALSE}$$

$$\text{Direct-Address-Search}(T, k)1 \quad \textbf{return } T[k]$$

Direct-Address Table (2)

- Complexity

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)\checkmark$

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$✓

So why isn't every set implemented with a direct-address table?

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$ ✓

  So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)$ ×
  - $|U|$ is typically a very large number—U is the *universe* of keys!
  - the represented set is typically *much smaller* than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$✓

  So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)\times$
  - $|U|$ is typically a very large number—U is the *universe* of keys!
  - the represented set is typically *much smaller* than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

- *Want: the benefits of a direct-address table but with a table of reasonable size.*

Direct Access Tables: Scorecard

Algorithm	*Complexity*
INSERT	$O(1)$✓
DELETE	$O(1)$✓
SEARCH	$O(1)$✓
MEMORY:	$\theta(M)\times$

Linked Lists

- *Interface*
  - LIST-INSERT(L, x) adds element x at beginning of a list L
  - LIST-DELETE(L, x) removes element x from a list L
  - LIST-SEARCH(L, k) finds an element whose key is k in a list L

Linked Lists

- *Interface*
  - LIST-INSERT(L, x) adds element x at beginning of a list L
  - LIST-DELETE(L, x) removes element x from a list L
  - LIST-SEARCH(L, k) finds an element whose key is k in a list L
- *Implementation*
  - a *doubly-linked* list
  - each element x has two "links" $x.prev$ and $x.next$ to the previous and next elements, respectively
  - each element x holds a key $x.key$
  - it is convenient to have a dummy "sentinel" element $L.nil$

Linked List With a “Sentinel”

LIST-INIT(L)

$L.nil.prev = L.nil$
$L.nil.next = L.nil$

LIST-INSERT(L, x)

$x.next = L.nil.next$
$L.nil.next.prev = x$
$L.nil.next = x$
$x.prev = L.nil$

LIST-SEARCH(L, k)

$x = L.nil.next$
while $x \neq L.nil \wedge x.key \neq k$
    $x = x.next$
return x