Unit #0: Introduction

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi-Fordoei[1]

2016W2: January-April 2017

[1] Thanks to Steve Wolfman for the content of most of these slides with additional material over the years 2004-2016 from Will Evans, Alan Hu, Ed Knorr, and Kim Voll.

Unit Outline

- Course logistics
- Course overview
- Fibonacci Fun
- Arrays
- Queues
- Stacks
- Deques

Course Information

Instructors

Anthony Estey	`avestey@cs.ubc.ca`
Ed Knorr	`knorr@cs.ubc.ca`
Mehrdad Oveisi-Fordoei	`moveisi@cs.ubc.ca`

TAs

Anthony Chen, Nancy Chen, Calvin Cheng, Harman Gakhal
Alistair Hackett, Arabelle Hou, Bibek Kaur, Qian Luo
Shahriar Noroozi Zadeh, Dejan Posavljak, Patience Shyu
Brian Tai, Vincent Tang, David Yin, May Young
Michael Zhang, David Zheng, Edward Zhou

Office hours

See `www.ugrad.cs.ubc.ca/~cs221`

Textbooks

- Susanna Epp's *Discrete Mathematics with Applications*
- Elliot Koffman and Paul Wolfgang's *Objects, Abstraction, Data Structures and Design Using C++*

Course Work

No late work; but we can exercise some discretion for medical cases, etc.

10%	Labs
15%	Programming projects (~ 3)
15%	Written homework (~ 3)
20%	Midterm exam
40%	Final exam

You must pass the final exam and the combination of labs/assignments in order to pass the course.

Collaboration

You may work in groups of two people on:

- Labs
- Programming projects
- Written homework

You may also collaborate with others as long as you follow the rules (see the website) and **acknowledge** their help on your assignment.

Don't violate the collaboration policy.

In other words, DON'T CHEAT!

Course Mechanics

- Web page: `www.ugrad.cs.ubc.ca/~cs221`
- Piazza:
  `https://piazza.com/ubc.ca/winterterm22016/cpsc221/home`
- UBC Connect site: `www.connect.ubc.ca`
- Most/all labs are in ICCS 015 (check your own timetable)
  - Use the `Xshell` program on the lab machines to `ssh` into a undergrad Unix machine (e.g. `lulu.ugrad.cs.ubc.ca`)
- Programming projects will be graded on `UNIX/g++`

What is a Data Structure?

Observation

- All programs manipulate data.
  - Programs process, store, display, and gather data.
  - Data can be information, numbers, images, sound, etc.
- The programmer must decide how to store and manipulate data.
- This choice influences the program in many ways:
  - Execution speed
  - Memory requirements
  - Maintenance (debugging, extending, etc.)

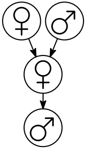
Goals of the Course

- Become familiar with some of the fundamental data structures and algorithms in computer science.
  - Learn when to use them.
- Improve your ability to solve problems abstractly.
  - Data structures and algorithms are the building blocks.
- Improve your ability to analyze algorithms.
  - Prove correctness.
  - Gauge, compare, and improve time and space complexity.
- Become modestly skilled with C++ and UNIX, but this is largely on your own!

Analysis Example: Fibonacci Numbers

Bee Ancestory:

1. Fertilized egg becomes a female bee with two parents
2. Unfertilized egg becomes a male bee with one parent



How many great-grandparents does a male bee have? great-great-grandparents? ...

Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

First two numbers are 1; each succeeding number is the sum of the previous two numbers.

Recursive Fibonacci

Problem: Calculate the nth Fibonacci number.

Recursive definition:

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

C++ code:

```
int fib(int n) {
   if (n <= 2)
       return 1;
   return fib(n-1) + fib(n-2);
}
```

Too slow! Why?

Iterative Fibonacci

Idea: Use an array

```
int fib(int n) {
  int F[n+1];

  F[0]=0; F[1]=1; F[2]=1;
  for( int i=3; i<=n; ++i ) {
    F[i] = F[i-1] + F[i-2];
  }
  return F[n];
}
```

(We don't really need the array.)

Can we do better?

Fibonacci by Formula

Idea: Use a formula (a *closed form solution* to the recursive definition.)

$$fib_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

where $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$.

```
#include <cmath>
int fib(int n) {
   double phi = (1 + sqrt(5))/2;
   return (pow(phi, n) - pow(-phi,-n))/sqrt(5);
}
```

Sadly, it's impossible to represent $\sqrt{5}$ exactly on a digital computer.

Can we do better?

Fibonacci with Matrix Multiplication

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix} = \begin{bmatrix}1+1\\ 1\end{bmatrix} = \begin{bmatrix}fib_3\\ fib_2\end{bmatrix}$$

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}2\\ 1\end{bmatrix} = \begin{bmatrix}fib_4\\ fib_3\end{bmatrix}$$

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}\begin{bmatrix}1\\ 1\end{bmatrix} = \begin{bmatrix}fib_n\\ fib_{n-1}\end{bmatrix}$$

How do we calculate $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}$?

Repeated Squaring

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{aligned} A \times A &= A^2 \\ A^2 \times A^2 &= A^4 \\ A^4 \times A^4 &= A^8 \\ A^8 \times A^8 &= A^{16} \\ A^{16} \times A^{16} &= A^{32} \\ &\vdots \end{aligned}$$

Example: $A^{100} = A^{64} \times A^{32} \times A^4$. 8 instead of 99 multiplications.
Generally, about $\log_2 n$ multiplications.

Is this better than iterative Fibonacci?

Abstract Data Type

Abstract Data Type

Mathematical description of an object and the set of operations on the object

Example: **Dictionary ADT**

- Stores pairs of strings: (word, definition)
- Operations:
  - Insert(word,definition)
  - Delete(word)
  - Find(word)

Another Example: Array ADT

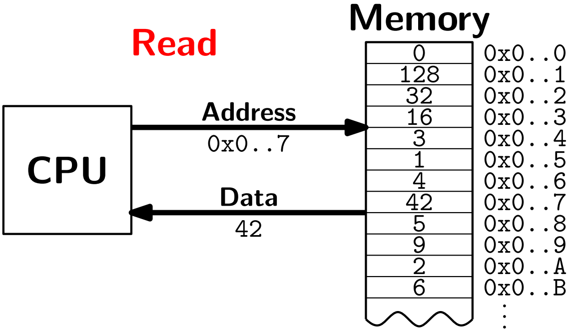
- Store things like integers, (pointers to) strings, etc.
- Operations:
  - Initialize an empty array that can hold n things.

    ```
    thing A[n];
    ```
  - Access (read or write) the ith thing in the array ($0 \leq i \leq n-1$).

    ```
    thing1 = A[i]; Read
    A[i] = thing2; Write
    ```

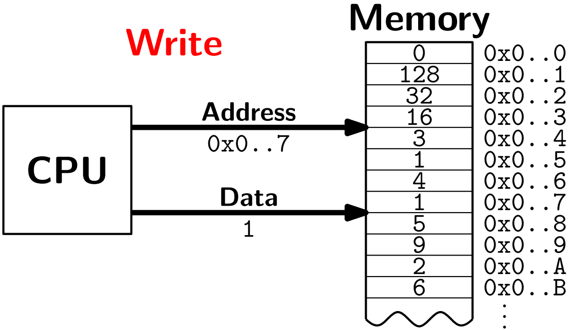
Why Arrays?

- Computer memory is an array.
  Read: CPU provides address i,
  memory unit returns the data stored at i.



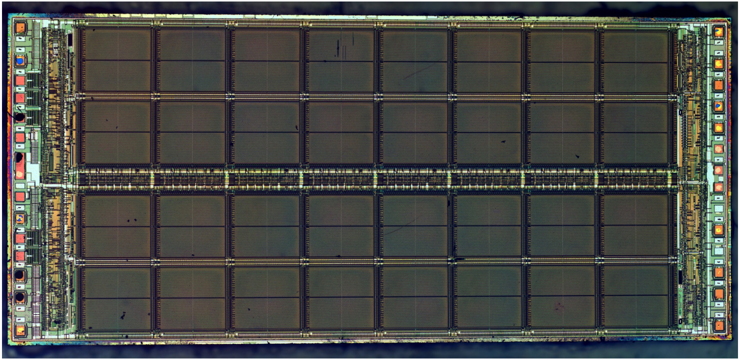
Why Arrays?

- Computer memory is an array.
  Write: CPU provides address i and data d,
  memory unit stores data d at i.



Why Arrays?

- Computer memory is an array. Every bit has a physical location.



Why Arrays?

- Computer memory is an array.
- Simple and fast.
- Used in almost every program.
- Used to implement other data structures.

Array Limitations

- We need to know the size of the whole array when the array is created.

  Fix: Resizeable arrays.
  If the array fills up, allocate a new, bigger array and copy the old contents to the new array. Then, delete the old array.

- Indices are integers 0,1,2,...

  Fix: Hashing. This will give us greater flexibility.
  (more later)

How Would You Implement the Array ADT?

How Would You Implement the Array ADT?

Arrays in C++

To Create:

```
int A[100];
```

To Access:

```
for ( int i=0; i<100; i++ )
    A[i] = (i+1) * A[i-1];
```

How Would You Implement the Array ADT?

Arrays in C++

To Create:

```
int A[100];
```

To Access:

```
for ( int i=0; i<100; i++ )
     A[i] = (i+1) * A[i-1];
```

Warning No bounds checking!

Data Structures as Algorithms; Abstract Data Types

Algorithm

- An algorithm is a high-level, language independent description of a step-by-step process for solving a problem.
- There may be multiple algorithms for solving a problem, and some may be more efficient than others.

Data Structure

- A data structure provides a way of storing and organizing data so that it can be manipulated by an ADT.
- An ADT describes *what* is stored, and it defines the *interface* (set of operations).
- An ADT is implemented by a data structure which specifies *how* the data is stored, and it provides *algorithms*
- An ADT may use different data structures in its implementation, for each operation.

Why So Many Data Structures?

Ideal Data Structure

- Fast, elegant, memory efficient

Trade-offs

- Time vs. space
- Performance vs. elegance
- Generality vs. simplicity
- One operation's performance vs. another's

Example: Data Structures for a Dictionary ADT

- List
- Skip list
- Binary search tree
- AVL tree
- Splay tree
- B-tree
- Red-Black tree
- Hash table

...

Code Implementation for a Dictionary

Theory

- An abstract base class (interface) describes the ADT.
- Descendents implement the data structures for the ADT.
- Data structures can change without affecting the client code.

Practice

- Different implementations sometimes suggest different interfaces (generality vs. simplicity).
- The performance of a data structure may influence the form of the client code (time vs. space, one operation vs. another).

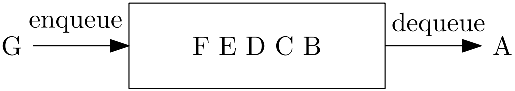
ADT Presentation Algorithm

1. Present an ADT.
2. Motivate it using some applications.
3. Repeat
   3.1 Develop a data structure for the ADT.
   3.2 Analyze its properties:
      - Efficiency
      - Correctness
      - Limitations
      - Ease of programming
4. Contrast the data structure's strengths and weaknesses.
   - Understand when to use each one.

Queue ADT

Queue operations

- create
- destroy
- enqueue
- dequeue
- is_empty



Queue property

If x is enqueued before y is enqueued, then x will be dequeued before y is dequeued.

FIFO: First In First Out

Applications of a Queue (Q)

- Holding jobs for a printer
- Storing packets on network routers
- Holding memory "freelists"
- Making wait lists fair
- Performing a breadth-first search (BFS)

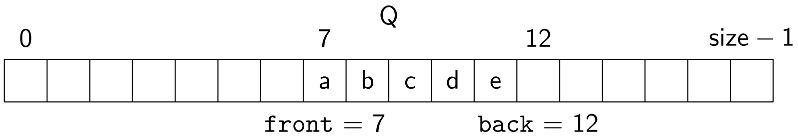
Abstract Q Example

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

In order, what letters are dequeued?

a. OATE
b. ROTA
c. OTAE
d. None of these, but it **can** be determined from just the ADT.
e. None of these, and it **cannot** be determined from just the ADT.

Circular Array Q Data Structure




```
void enqueue(Object x) {
  Q[back] = x;
  back = (back + 1) % size;
}

bool is_empty() {
  return (front == back);
}
```

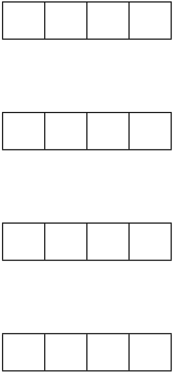
```
Object dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}

bool is_full() {
  return (front ==
     (back + 1) % size);
}
```

Circular Array Q Example

front back

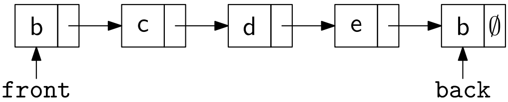
enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue



What are the final contents of the array?

a. RTE
b. RTET
c. TETA
d. TE
e. None

Linked List Q Data Structure




```
void enqueue(Object x) {
  if (is_empty())
    front = back = new Node(x);
  else {
    back->next = new Node(x);
    back = back->next;
  }
}

bool is_empty() {
  return (front == NULL);
}
```

```
Object dequeue() {
  assert(!is_empty());
  Object ret = front->data;
  Node *temp = front;
  front = front->next;
  delete temp;
  return ret;
}
```

DIY memory management

Compare: Circular Array vs. Linked List

Ease of implementation

Generality

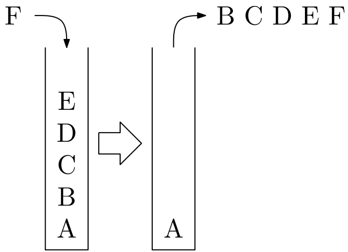
Speed

Memory use

Stack ADT

Stack operations

- create
- destroy
- push
- pop
- top
- is_empty



Stack property

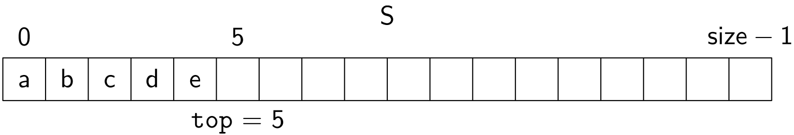
if x is pushed before y is pushed, then x will be popped after y is popped.

LIFO: Last In First Out

Stacks in Practice

- Implementing a function call stack
- Removing recursion
- Balancing symbols (e.g., parentheses)
- Evaluating Reverse Polish Notation (RPN)
- Performing a depth-first search (DFS)

Array Stack Data Structure

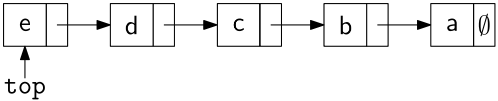



```
void push(Object x) {
  assert(!is_full());
  S[top] = x;
  top++;
}

Object top() {
  assert(!is_empty());
  return S[top-1];
}
```

```
Object pop() {
  assert(!is_empty());
  top--;
  return S[top];
}
bool is_empty() {
  return( top == 0 );
}
bool is_full() {
  return( top == size);
}
```

Linked List Stack Data Structure




```
void push(Object x) {
  Node *temp = top;
  top = new Node(x);
  top->next = temp;
}

Object top() {
  assert(!is_empty());
  return top->data;
}
```

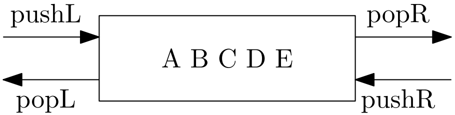
```
Object pop() {
  assert(!is_empty());
  Object ret = top->data;
  Node *temp = top;
  top = top->next;
  delete temp;
  return ret;
}

bool is_empty() {
  return( top == NULL );
}
```

Deque ADT

Deque (Double-ended queue) operations

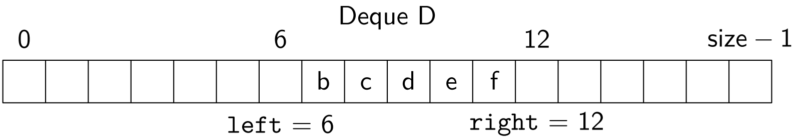
- create/destroy
- pushL/pushR
- popL/popR
- is_empty



Deque property

Deque maintains a list of items.
push/pop adds to/removes from front(L)/back(R) of list.

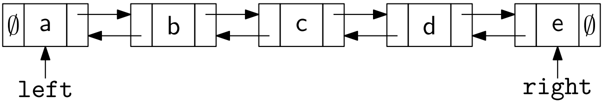
Circular Array Deque Data Structure




```
void pushL(Object x) {
  assert(!is_full());
  D[left] = x;
  left = (left - 1) % size;
}
Object popR() {
  assert(!is_empty());
  right = (right - 1) % size;
  return D[right];
}
...
```

```
bool is_empty() {
  return( left ==
      (right - 1) % size);
}
bool is_full() {
  return( left ==
      (right + 1) % size);
}
```

Linked List Deque Data Structure




```
void pushL(Object x) {
  if ( is_empty() )
    left = right = new Node(x);
  else {
    left->prev = new Node(x);
    left->prev->next = left;
    left = left->prev;
  }
}

bool is_empty() {
   return left==NULL;
}
```

```
Object popR() {
  assert(!is_empty());
  Object ret = right->data;
  Node *temp = right;
  right = right->prev;
  if (right) right->next = NULL;
  else left = NULL;
  delete temp;
  return ret;
}
```

Data Structures You Should Already Know (Somewhat)

- Arrays
- Linked lists
- Trees
- Queues
- Stacks