Computer Systems 1
Lecture 17

Trees

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

1. Survey about programming in Python
2. Assessed exercise: ordered lists program
3. Trees

Survey about programming in Python

- This is completely optional
- There is a research group investigating the process of learning a programming language
- We are running two surveys
  1. Starting Tuesday 5 March at 1pm, closing Tuesday 12 March at 12:00 noon
  2. The lecture on Tuesday 12 March will be about programming language semantics: relating Python to compilation patterns and machine language
  3. Followup survey starting Tuesday 12 March at 1pm, closing Tuesday 19 March at 12:00 noon
  4. The last lecture of the course is Friday 21 March, and will discuss the results
- Participation is optional and anonymous.
- It will not affect your grade in any way, but we hope you find it interesting and helpful in learning programming languages

Assessed exercise: ordered lists program

1. There is one lab exercise that will be assessed: it counts for 10% of your grade in the course
2. It will be posted tonight on Moodle
3. You are given a reasonably long program, which contains a few small missing pieces
4. The exercise is to
    1. Read and understand the program
    2. Complete the missing pieces

Concepts used in the program

1. Array of records
    1. Representing a command as a records
    2. Traversing an array of records
    3. Case statement and jump table
2. Linked lists
    1. Traversing a list to print its elements
    2. Insertion in list keeping the elements in ascending order
    3. Deletion from a list
    4. Searching a list

Ordered lists

There is an array of lists, initially empty. There are nlists of them.

```
list[0] = [ ]
list[1] = [ ]
...
list[nlists-1] = [ ]
```

At all times as the program runs, the lists are ordered: their elements are increasing

```
list[0] = [4, 9, 23, 51 ]
list[1] = [7, 102, 238 ]
...
list[nlists-1] = [2, 87, 89, 93, 103, 195 ]
```

Commands

The program executes commands:

- Terminate — the program finishes
- Insert into list i the value x — modify list[i] so it contains x, while maintaining the ascending order
- Delete from list i the value x — modify list[i] so x is removed, but don't do anything if x isn't in the list
- Search list i for x — print Yes if x is in the list, No otherwise
- Print i — the numbers in list[i] are printed

Example

- Insert into list[3] the value 23 [23]
- Insert into list[3]the value 6 [6, 23]
- Insert into list[3] the value 67 [6, 23, 67]
- Insert into list[3] the value 19 [6, 19, 23, 67]
- Print list[3] 6 19 23 67

Why are ordered lists useful?

- This is one way to arrange a database: think of the elements as persons' names, or matriculation numbers
- Sometimes you want to process all the data in a container in a specified order
- If the data is ordered, it's faster to find a particular item (on average you only have to check half of the items)
- An ordered list can be used to represent a set

Where do the commands come from?

- In a real application, we would read the commands from input
- But in this program, each command is represented as a record
- The entire input is a static array of records defined with `data` statements
- This is easier because
  - If you read from an input device, it's necessary to convert the input character string to numbers
  - In testing a program, it's convenient to have input data that is fixed and repeatable
  - Don't want to have to type in the same input every time you run the program!

Representing a command

- Each command is a record with three fields
  - A code indicating which kind of command
  - A number i indicating which list we're operating on
  - A value x which might be inserted etc
- Each record must have these three fields
- Some commands don't use them all (e.g. Print just needs i, not x)
- The main program uses a case statement to handle each command, and implements this with a jump table

Reading a program before writing

- You should *read and understand* the program before modifying it
  - Reading a program is an important skill you will need throughout your career
  - The program is filled with examples so it is excellent revision material
  - You need to understand a program before you'll be able to make changes to it
- One of the aims of the exercise is to get experience with reading a longer program—don't skip this!

Some tips on testing and debugging

- Debugging has two phases:
  1. Diagnosis: finding out what went wrong and why
  2. Correction: fixing the error
- The most important point: don't just make random changes to the code and hope for the best—instead, find out what the error is and fix it cleanly

Reading and testing a program

- A good way to understand a section of assembly language instructions is to step through it, one instruction at a time
  1. Check that the instruction did what you expected it to do
  2. Check that the instruction is consistent with its comment
  3. Try to relate the instruction with the bigger picture: what is it doing in the context of the program?
- Coverage
  1. You don't need to step through a set of instructions a huge number of times
  2. If there's a loop, step through two or three iterations
  3. If possible, arrange test data so the loop will terminate after just a few iterations
  4. But try to step through as much of the program as possible
  5. This is called coverage: try to cover all of the program with your testing

Breakpoints

- It's a good idea to step through a program one instruction at a time, so you understand clearly what each instruction is doing
- However, in a longer program this isn't always feasible
    - The OrderedLists program has to build the heap when it starts; this may take several thousand instructions before it even really gets going!
- Solution: breakpoints
    - Find the address of an instruction where you want to start single stepping
    - Enter this address as a breakpoint
    - Click Run to execute the program at full speed; when it reaches the breakpoint it will stop
    - Then you can single step to examine what the instructions are doing

How to set a breakpoint

- On the Processor pane, click Breakpoint. It will say "Breakpoint is off"
- Enter the breakpoint command and click Set Breakpoint
- `BPeq BPpc (BPhex "01a6")`
- It will say "Breakpoint is on". Click Close
- On Processor, click Run. It will stop when the pc register gets the value you specified

Tree

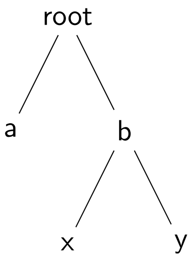
- A node doesn't have to have two fields named *value* and *next* — it's normal to define a specific node type for an application program.
- Nodes with *value* and *next* can be connected into a linked list.
- Nodes can also have with several fields containing data, not just one "value" field.
- And a node can have several pointer fields...
- Common case: a binary tree has two pointers in each node, named left and right.
- Each of these can either contain nil, or point to another node.

Tree

```
Node : record
  value    ; the actual data in the node
  left     ; left subtree is a pointer to a Node
  right   ;  right subtree is a pointer to a Node
```

- Similar to a node for a linked list, but with two pointers
- There can also be several fields for data, not just one “value” field
- And we could have more than just two pointers

A binary tree



In computer science, for some reason we draw trees upside down

Suppose p is a pointer to the tree

- (*p).left is the pointer to the left subtree
- (*p).right is the pointer to the right subtree

Applications of trees

Trees are used everywhere in programming

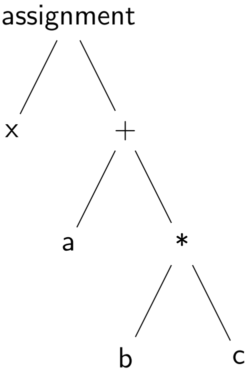
- To hold structured data
- To make programs faster (*much* faster)

Holding structured data

- A compiler reads in program text, which is just a character string: a sequence of characters.
- It needs to represent the deep structure underlying that sequence of characters.
- This is done by building a tree (the part of a compiler that takes a character string and produces a tree is called the parser).

Parsing

```
x := a + b * c
```



Another application of jump tables!

- In complicated applications, trees normally have several different types of node
- Examples: operations with 1 operand; operations with 2 operands; control constructs with a boolean expression and two statements, etc.
- So there are several different kinds of record
- Each record has a *code* in the first word
- The value of the code determines how many more words there are in the record, and what they mean
- When a program has a pointer to a node, it needs to examine the code and take different actions depending on what the code is
- This is done with a jump table

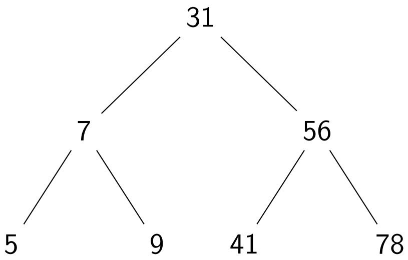
Searching

- Suppose we have a large number of records (e.g. a database)
- We want to search the database for an entry where a field has a certain value (e.g. search for a record where the MatricNumber field is 123456)
- If you just have these records in an array, or a linked list, you have to search them one by one
- On average, you have to look at half the entries in the database to find the one you want
- If you double the size of the database, you double the average time to look up an entry
- Terminology: this is called *linear time* or $O(n)$ complexity

A better approach

- Linear search is silly if you can place the records in order
- You're trying to find the telephone number of John Smith in the phone book
- Would you do this?
  1. It isn't Aardvark, Aaron
  2. It isn't Acton, Rebecca
  3. It isn't Anderson, Susan
  4. It isn't Atwater, James
  5. . . . 8 million more unsuccsessful searches because this is the Los Angeles directory
- That's silly!
- Open the book to the middle, notice that S is in the second half
- Open the book to the middle of the second half . . .
- Each time you look at an entry in the book, you discard half of the remaining possibilities

Binary search tree



- At every level: if a node contains x, then
  - every node in the left subtree is less than x, and
  - every node in the right subtree is greater than x.
- You can search the tree by starting at the root, and at every step you *know* whether to go left or right

Algorithmic Complexity

- Complexity is concerned with how the execution time grows as the size of the input grows
- This is expressed as a function of the input size n
- Normally we don't care about the *exact* function, and we use O-notation. Instead of a funciton like $f(n) = 4.823 \times n$, we just write $f(n) = O(n)$
  - $O(1)$ — if input grows, the execution time remains unchanged. This is unrealistic: the program cannot even look at the input!
  - $O(n)$ — if the input is 5 times bigger, the execution time is 5 times bigger. This is the best you can hope for
  - $O(n^2)$ — if the input is 5 times bigger, the time is 25 times bigger

Algorithm is more important than small optimisation

- Some programmers spend lots of effort trying to save one or two instructions in a piece of a program
  - But it doesn't matter much whether a program takes 2.00032 seconds or 2.00031 seconds
- It's much more important to use a suitable algorithm
  - On small data it doesn't make much differnce
  - On large (realistic) data, a better algorithm makes a huge difference

Complexity for search

- Ordered lists
  - The Ordered Lists program has an operation to search a list for a value x
  - On average, you need to look through half of the data to find out whether x is present
  - If the list were *not* ordered, you would need to look through *all* of the data to determine whether x is present
  - So the ordered list makes the search about twice as fast
  - But in either case, this is $O(n)$ — if you double the data size, the average time is doubled
- Binary search tree
  - The number of comparisons needed is roughly the height of the tree
  - If the tree is *balanced*, the time complexity is $O(\log n)$

How much faster?

- With a linear data structure (array, linked list)
  - Each time you compare a database entry with your key, you eliminate one possibility
  - The time is proportional to the size of the database
  - It's called *linear time* — time $= O(n)$
  - For 2 million records, you need a million comparisons
- With a binary search tree
  - Each time you compare a database entry with your key, you eliminate (on average) half of the possibilities
  - The time is proportional to *the logarithm of the size* of the database
  - It's called *log time* — time $= O(\log n)$
  - For 2 million records, you need 21 comparisons
  - There's a saying: "logs come from trees"

A common pitfall

- When you're writing a program, it's natural to test it with small data
- Even if the algorithm has bad complexity, the testing may be fast
- But then, when you run the program on real data, the execution time is intolerable
- That means going back and starting over again
- So it's a good idea to be aware of the complexity of your algorithm from the beginning

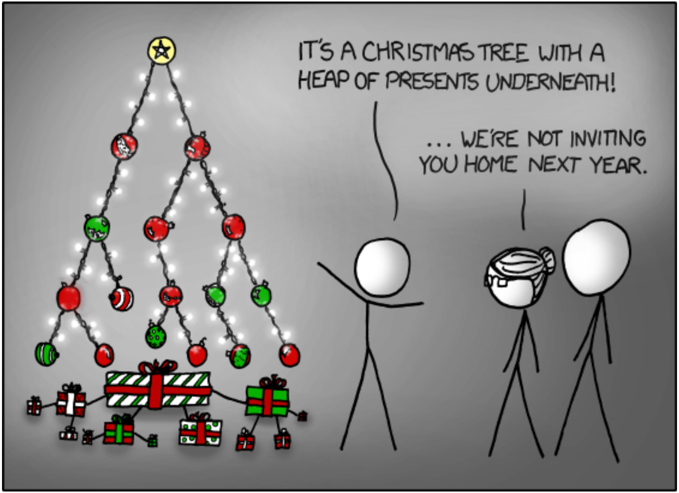
How bad can complexity be?

Order of magnitude estimate of time for input of size n

n	$\log n$	$n \log n$	n^2	2^n
1	1	1	1	2
10	3	30	100	1,000
100	7	700	10,000	1267650600228229401496703205376
1,000	10	10,000	1,000,000	$>$ age of universe

- Lots of real problems have data size larger than 1,000
- Lots of algorithms have exponential complexity: 2^n

tree



https://xkcd.com/835/