CSE100

Advanced Data Structures

Lecture 1

(Based on Paul Kube course materials)
CSE 100

- Overview of course requirements
- Outline of CSE 100 topics
- Review of trees
- Helpful hints for team programming
CSE 100 course page

All information related to the course is available in the textbook or online, following links from the class home page:

http://cseweb.ucsd.edu/classes/fa15/cse100-a

You’re responsible for knowing that information, so make a note of that URL and read what’s there
Topics for the course

• In CSE 100, we will build on what you have already learned about programming: procedural and data abstraction, object-oriented programming, and elementary data structures and algorithm design, implementation, and analysis
• We will build on that, and go beyond it, to learn about more advanced, high performance data structures and algorithms:
  • Balanced search trees: AVL, red-black, B-trees
  • Binary tries and Huffman codes for compression
  • Graphs as data structures, and graph algorithms
  • Data structures for disjoint-subset and union-find algorithms
  • Hash tables, hash functions, and hashing techniques
  • Randomized data structures: skip lists, treaps
  • Amortized cost analysis
  • The C++ standard template library (STL)
Data structures

- A data structure is... a structure that holds data
- A data structure is an object that offers certain useful operations through its “Application Programmer Interface” (API), for example: storing, retrieving, and deleting data of a certain type
- A data structure may offer certain performance guarantees on its operations, for example certain best-, worst-, or average-case time or space costs
  - To meet those performance guarantees, a data structure may need to be implemented in a particular way
- In CSE 100 we will study the performance guarantees that are permitted by various data structure implementations
- We will begin by reviewing trees...
A review of trees

- A tree is a hierarchical (not just linear, and not unstructured!) data structure
- A tree is a set of elements called nodes, structured by a "parent" relation:
  - If the tree is nonempty, exactly one node in the set is the root of the tree
  - The root of a tree is the unique node that has no parent
  - Every node in the set except the root has exactly one other node that is its parent
• The root goes at the top: here node A is the root of the tree (in Computerscienceland, trees grow upside down)
• The parent of a node is drawn above that node, with a "link" or "edge" from the node to its parent: here node A is the parent of nodes B,C ; and B,C are called the children of A
• Some nodes have no children, and are called leaves of the tree: here nodes D, G, I, J, L are leaves
Tree terminology: definitions

- **Children** of a node P: the set of nodes that have P as parent
- **Descendant** of a node P:
  - If a node C is a child of P, then C is a descendant of P
  - If a node C is a child of a descendant of P, then C is a descendant of P
- **Ancestor** of a node C: if C is a descendant of P, then P is an ancestor of C
- **Root** of a tree: the unique node in the tree with no parent
- **Leaves** of a tree: the set of nodes with no children
- **Subtree**:
  - The empty tree is a subtree of every tree
  - Any node of a tree together with its descendants is a subtree of the tree
- **Level or depth** of a node (using ‘zero-based’ counting):
  - The level of the root is 0.
  - The level of any non-root node is 1 + the level of its parent (this is equal to the number of edges on the path from the root to the node)
- **Height** of a node: the height of a node is the number of edges on the longest path from the node to a leaf
- **Height** of a tree: the height of the root of the tree
Binary trees

• A binary tree is a tree in which every node has at most two children
• A particular child of a binary tree node is either a “left child” or a “right child”
• Recursive definition of "binary tree": either the empty tree, or a node together with left and right subtrees which are both binary trees
• Examples:
Important binary tree properties

• Consider a “completely filled” binary tree (every level that has any nodes at all has as many nodes as possible):
  • How many nodes at level 0?
  • How many nodes at level 1?
  • How many nodes at level 2?
  • How many nodes at level 3?

• Generalizing, how many nodes at level L? __________
• And so, how many nodes in a completely filled binary tree of height H? __________
• And so, what is the height of a completely filled binary tree with N nodes?________
In a completely filled binary tree with $N$ nodes...

- Generalizing, how many nodes at level $L$? $2^L$

- And so, how many nodes in a completely filled binary tree of height $H$?

$$N = \sum_{L=0}^{H} 2^L = 2^{H+1} - 1$$

- And so, what is the height of a completely filled binary tree with $N$ nodes?

$$2^{H+1} = N + 1$$
$$H + 1 = \log_2(N + 1), \text{ so}$$
$$H = O(\log N), \text{ and } H = \Omega(\log N), \text{ and so } H = \Theta(\log N)$$
Reviewing “big-O” notation

• Write:

\[ g(N) = O(f(N)) \]

• And say: " \( g(N) \) is ‘big-O’ of \( f(N) \)" if there are positive constants \( c, n_0 \) such that for all \( N \geq n_0 \)

\[ g(N) \leq cf(N) \]

... that is, \( g \) eventually grows no faster than \( f \) (times a constant).
... \( f \) gives an asymptotic upper bound on the rate of growth of \( g \).
... the order of \( g \) is at most the order of \( f \)
Reviewing “big-Ω” notation

• Write:

\[ g(N) = \Omega(f(N)) \]

• And say: "\( g(N) \) is ‘big-omega’ of \( f(N) \)" if there are positive constants \( c, n_0 \) such that for all \( N \geq n_0 \)

\[ g(N) \geq cf(N) \]

... that is, \( g \) eventually grows at least as fast as \( f \) (times a constant).

... \( f \) gives an asymptotic lower bound on the rate of growth of \( g \).

... the order of \( g \) is at least the order of \( f \)
Reviewing “big-Θ” notation

• Write:

\[ g(N) = \Theta(f(N)) \]

• And say: "\( g(N) \) is ‘big-theta’ of \( f(N) \)" if \( g(N) \) is both ‘big-O’ and ‘big-omega’ of \( f(N) \).

\( f \) gives a good qualitative estimate -- a “tight bound” -- on the rate of growth of \( g \)

... the order of \( g \) is the same as the order of \( f \)
Generalizing binary trees: K-ary trees

- An K-ary tree is a tree in which every node has at most $K$ children.
- $K=2$ gives binary trees, $K=3$ gives ternary trees, etc.
- Possible children of a node in a K-ary tree are sequentially ordered, left-to-right.
- Recursive definition of "K-ary tree": either the empty tree, or a node together with at most $K$ subtrees which are all K-ary trees.
- Examples of useful K-ary ($K>2$) trees we will cover later: 2-3 trees, B-trees, alphabet tries.
- Basic properties of binary trees generalize to properties of K-ary trees...
In a completely filled K-ary tree with M nodes...

• Generalizing, how many nodes at level L? \( K^L \)

• And so, how many nodes in a completely filled K-ary tree of height H?

\[
N = \sum_{L=0}^{H} K^L = \frac{K^{H+1} - 1}{K - 1}
\]

• And so, what is the height of a completely filled K-ary tree with N nodes?

\[
K^{H+1} = N(K - 1) + 1
\]

\[
H + 1 = \log_K(N(K - 1) + 1), \text{ so}
\]

\[
H = O(\log N), \text{ and } H = \Omega(\log N), \text{ and so } H = \Theta(\log N)
\]
Tree properties, continued

- A completely filled K-ary tree with N nodes has the minimum height possible of any K-ary tree with N nodes... In fact for any K-ary tree,

\[ H \geq \log_K (N(K - 1) + 1) - 1 \]

and so

\[ H \geq \lfloor \log_K N \rfloor - 2 \]

- But what is the maximum height possible for a K-ary tree with N nodes? __________
Tree properties, good and bad

• Many interesting operations on tree data structures have a time cost that, in the worst case, is proportional to the height of the tree
  • Find, insert, and delete operations in search trees,
  • Insert and delete-root operations in heaps, etc.

• For a completely filled tree, that means that these operations have a worst-case time cost that is a logarithmic function of the number of nodes N in the tree
  • log(N) grows very slowly as a function of N, which is good!
  • This fact is one of the main reasons that trees are an important data structure

• But for a “worst-case” tree, that means that these operations have a worst-case time cost that is a linear function of the number of nodes N in the tree
  • This means the time cost grows proportionally to N, which is not very good!
  • This fact is a major problem with using trees in many applications, but it can be overcome, as we will see
The importance of being balanced

- A binary tree of $N$ nodes is considered *balanced* if its height is “close to” $\log_2(N)$
- The usual simple algorithm for inserting nodes in a search tree can produce unbalanced trees, which lead to poor performance
  - ... and this is can easily happen in practice: for example, it will happen if the keys to be inserted are sorted or almost sorted
- With cleverer insert operations, you can make sure the tree is always balanced no matter what, and guarantee excellent worst-case performance... at the cost of a more complicated implementation
- We will first look at implementation issues for binary search trees in general
- Then later we will look at a few approaches to improving performance of binary search trees:
  - AVL trees, red-black trees, B-trees, splay trees, randomized search trees
Working in teams

• In CSE 100 this quarter, you are allowed and encouraged to write your programming assignments in teams of 2 (See the assignment README’s for details)

• This can work out very well, if members of the team have compatible skills, schedules, and personalities, and if you follow some basic software engineering principles (Otherwise, it won’t work well, and you will be better off working on your own)

• We will sketch two approaches that you can use:
  • The standard software life cycle
  • Extreme programming (More information is available on the web and elsewhere)
The standard software life cycle

- Some variant of this is used in most software projects. Basic steps:
  - Requirements
    - Get the requirements for the software from the customer (or in CSE 100 from the assignment README...)
    - Make sure they are clear and that you understand them!
  - Design
    - Before coding, create a design for a software system that will meet the requirements
    - Use good design principles: top-down decomposition, abstraction, modularity, information hiding, etc.
  - Code
    - Divide the coding task among members of the team, and code according to a common standard (including style and comments)
  - Test
    - Thoroughly test each “unit” (block, method, class, package) and the entire system
  - Deploy
    - Deliver the completed system (or in CSE 100, turn in your assignment)
Extreme programming ("XP")

- A new and somewhat different software engineering approach, very successful when used for small software projects (say, less than 16 programmers). Some XP practices:
  - The planning process
    - Rank desired system features by importance, determined by need and cost
  - Pair programming
    - Write all code in pairs, two programmers working together at one machine
    - “Driver” controls the keyboard and mouse and types code; “Observer” makes suggestions, identifies problems, thinks strategically. Both brainstorm as needed.
    - Switch roles often
  - Small releases
    - Put a simple system into production early, and update it frequently on a short cycle
  - Test first and often
    - XP teams focus on validation of the software at all times. Write unit tests first, then write code that fulfills the requirements reflected in the tests
  - Refactoring
    - XP teams improve the design of the system throughout the entire development. This is done by keeping the software clean: without duplication, simple yet complete
Basic software principles

Whether you use XP or a more traditional approach, some things to keep in mind:

• Top-down, divide-and-conquer design strategies are useful; break the overall problem down into small, manageable subproblems

• Using object-oriented design, think of solving the problem using objects that have certain properties (instance variables) and behaviors (instance methods)

• In your design, be clear about what is the interface to each piece of your solution

• In object-oriented design, the interface to a piece of the solution consists of the public (static or instance) methods of a class; be clear about PRE and POST conditions, and what arguments, return values, and side effects each method has

• With a good design, each member of the team can, in principle, work on the implementation of a piece of the solution separately; as long as the implementation meets the interface specification, the system will work

• A strategy: at first, use implementations that are fast and easy to write, but that work. Later, improve the implementation for better performance, while keeping the interface the same

• Be sure to test and debug your solution! This may lead to a redesign, but hopefully it involves only small changes to the implementation
Getting started

- Read Weiss chapter 1 for a review of some mathematical concepts, and an introduction to C++
- Read Lecture 2 slides for a comparison of Java to C++
- Read Weiss chapter 2 for a review of asymptotic analysis and big-O notation
- Read Weiss chapter 4 for a review of trees

- We will then start with a C++ implementation of a binary search tree class template
  
  ➔ See Assignment 1 as soon as possible
Next time

• An introduction to C++
• Comparing Java and C++
• Basic C++ programming
• C++ primitive types and operators
• Arrays, pointers and pointer arithmetic
• The interface/implementation distinction in C++
• C++ class templates
  Reading: Weiss Ch 1