

Great Ideas in Computing

Allan Borodin

University of Toronto CSC196
Fall 2026

Class 8: October 5 (2026)

Announcements and Today's Agenda

Announcements

I have posted the start of Assignment 2. It is due October 21.

Agenda

- Questions asked in the last tutorial.
- Your Comments on Guest Presentation.
- A short discussion on updating a linked list and a (balanced) binary search tree.
- A few basic graph theoretic definitions; search trees as a special case of *directed acyclic graph*.
- Hash Tables and the Birthday Paradox: another way to implement a Dictionary

Note: A basic understanding of graph concepts and probability is essential for Computer Science.

Accessing a linked list or search tree

I didn't explicitly state it, but for a linked list (respectively, a search tree), we need to be able to access the data structure by a link to the "head" of the linked list (respectively, the root of the search tree).

Inserting a new item into a linked list is very efficient (constant time). Just add the new element to be the head of the list and link the new head to the old head of the list.

For deleting an item, we need to first find the item (if it exists) in the list and then rearrange the links as I will illustrate. After finding the item, this is also constant time to delete.

For search trees, the process for inserting a new record (i.e., inserting a new key) requires first finding out where it belongs and then we have to readjust links to add new key. If we want to maintain the tree being binary and balanced, we will need $O(\log n)$ time to rebalance the tree. Same for deletions.

Search trees as a special case of graphs/networks

Let's again consider the search tree data structure which I will again illustrate on the white board.

In discussing search trees, we will use the terminology of graph theory so as to make definitions precise. Lots of notation and definitions but hopefully the discussion will make things clear. A search tree is labelled rooted tree which is a special case of a **directed acyclic graph (DAG)**.

A directed graph (also called a directed network) $G = (V, E)$ consists of a set of **nodes** (also called vertices) V and a set of edges E (also called links) where each edge is an ordered pair $\langle u, v \rangle$ of nodes. We are drawing a representation of an edge on the board. **Note:** We will usually be assuming **simple** directed graphs where $u \neq v$ (i.e., no self loops) and no multiple edgess between two vertices.

A graph (also called an undirected graph) is almost formally the same but now edges are unordered pairs $\{u, v\}$. We will often just denote a directed or undirected edge as (u, v) when the context is clear.

Note: See Easley and Kleinberg text or my CSC303 slides for examples.

A few more graph theoretic definitions

For directed graphs, the **in-degree** of a node v in G is $|\{(u, v) \in E\}|$; that is, the number of edges into v . The out-degree of a node v is $|\{(v, z) \in E\}|$; that is the number of edges leaving v .

A directed or undirected path π from v to z in a directed graph is a sequence of edges $(u_0, u_1), (u_1, u_2), \dots, (u_{\ell-1}, u_{\ell})$ with $v = u_0$ and $z = u_{\ell}$. The path length of π is ℓ . We are only considering simple paths such that $u_i \neq u_j$ for $i \neq j$.

A directed tree is a special case of a directed graph with the following properties:

- There is a unique node (called the **root** having in-degree 0 ; lets call the root r .
- For every node $v \neq r$, there is a unique path from r to v .

It follows that every node $v \neq r$ has in-degree 1.

The **depth**(v) of a node v in a directed tree (or a DAG) is the length of the unique path from r to v . (The root is at depth 0.)

A leaf in a directed tree is a node having out-degree 0.

Search trees as labelled directed trees; binary search trees

In a graph or directed graph we may want to label the nodes and/or edges.

A search tree is a directed tree where the nodes are labelled by record identifiers (also called “keys”). We could then use pointers to the record corresponding to the key (or if the records don’t contain much information, then store the entire record in a node).

For simplicity, we will only consider *binary search trees* where every node has out-degree at most 2. (In a *strict binary tree*, every non-leaf node has out-degree exactly 2.)

In a binary search tree, the left (resp. right) “subtree” of a node v consists of all the nodes whose identifiers (i.e. labels) are less than (resp. greater than) the identifier of v . (For a Dictionary, We can assume that there are no identical keys.)

Final comments on binary search trees.

We can make every binary search tree into a strictly binary search by adding a “dummy” leaf for any missing edge. A dummy leaf indicates that there is no key at this leaf which satisfies the definition of a binary search tree. So let's restrict our focus to strictly binary search trees.

A (strictly) binary search tree is *balanced* if there is some d such that the path length to every leaf is d or $d + 1$. It follows that in a balanced strictly binary search tree with n vertices, where $2^k - 1 \leq n < 2^{k+1} - 1$, the depth of every leaf node is either $k - 1$ or k . It is sufficient for our purposes to have nearly balanced search trees with depth $d \approx \log(n)$.

In a balanced binary search tree with n keys, a record can be accessed (or known to be missing) search time $O(\log n)$ I hope this is easy to see as this is like binary search.

What is not so easy to see is that a balanced binary search tree can be re-balanced after an update or an insertion/deletion in $O(\log_2 n)$ time. So balanced search trees seem like a good compromise for dynamic dictionaries.

Final comments on binary search trees.

We can make every binary search tree into a strictly binary search by adding a “dummy” leaf for any missing edge. A dummy leaf indicates that there is no key at this leaf which satisfies the definition of a binary search tree. So let's restrict our focus to strictly binary search trees.

A (strictly) binary search tree is *balanced* if there is some d such that the path length to every leaf is d or $d + 1$. It follows that in a balanced strictly binary search tree with n vertices, where $2^k - 1 \leq n < 2^{k+1} - 1$, the depth of every leaf node is either $k - 1$ or k . It is sufficient for our purposes to have nearly balanced search trees with depth $d \approx \log(n)$.

In a balanced binary search tree with n keys, a record can be accessed (or known to be missing) search time $O(\log n)$ I hope this is easy to see as this is like binary search.

What is not so easy to see is that a balanced binary search tree can be re-balanced after an update or an insertion/deletion in $O(\log_2 n)$ time. So balanced search trees seem like a good compromise for dynamic dictionaries.

Everything you need to know *now* about graphs

See <http://www.cs.toronto.edu/~bor/303s20> which is the home page for the 2020 spring version of CSC303 (Social and Information Networks). There you will find a link to the course contents. Take a look at the slides for week 1 for all the graph concepts you will need for this course and probably for most future courses.

Slides 33-39 have the basic definitions for undirected graphs. Slides 45-47 briefly discuss directed graphs and weighted graphs. Slide 56 illustrates a bipartite graph and mentions an interesting difference between a graph being *2-colourable* and being *3-colourable*.

You will also find many examples of applications of graphs (also called networks) and directed graphs. Slides 8-23 in the slides for week 1 of CSC303, spring 2020 have examples of networks.

Graphs (and graph algorithms) are used throughout CS and many other disciplines. Getting familiar with these concepts is important.