

Great Ideas in Computing

Allan Borodin

University of Toronto CSC196
Fall 2026

Class 6: September 28 (2026)

Announcements and Today's Agenda

Announcements

- Reminder: Assignment 1 is due this Wednesday at 2:30PM. Please let me know if there is any difficulty in submitting. The late policy is a 5% deduction for each day of lateness up to two days.
- Reminder: Professor Peter Love will give a guest presentation this Wednesday. The topic is quantum computation.
- Slides for L5 are now posted. I also fixed some typos in L3 and L4. I very much appreciate corrections and suggestions for improving the slides.

Agenda

- Data structures for the Dictionary data type.
- A table of complexity time bounds.
- Very brief introduction to graph theory.
- Hash tables and the birthday paradox

Returning to the Dictionary Data Type and different data structures for implementing a dictionary

We ended the last class with a quick overview of four of the five different data structures for the dictionary data type. I am repeating these data structures here and then we will elaborate more on these 5 different data structures in terms of their pros and cons. The same data structures could be used for other data types, such as a *priority queue* but would not be as efficient for the main use of a priority queue.

Here are some well known ways (called *data structures*) to implement a dictionary.

- ❶ An unordered list in an array
- ❷ An ordered list in an array
- ❸ A linked list
- ❹ A (balanced) search tree.
- ❺ A hash table.

Brief Discussion on these Different Methods

I just want to give a very high level idea of these different ways to implement a dynamic dictionary mentioning some tradeoffs and introducing some related concepts.

I will assume that we know n the size of the dictionary (i.e. how many records) or at least a reasonable estimate of the size. Otherwise we have to consider how to dynamically allocate memory as needed. There are enough concepts to consider even without worrying about dynamic memory management.

Each item has a unique name or *identifier*.

After I describe each method (on the board), let's discuss some pros and cons of each method.

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array and don't worry about words missing an entry after a deletion.)

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array and don't worry about words missing an entry after a deletion.)

Requires an “average” of $n/2$ comparisons to find a current item and n comparisons to determine if the requested item is not in the current array. This is a hint of an important issue: namely, **what does *average* mean?**

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array and don't worry about words missing an entry after a deletion.)

Requires an “average” of $n/2$ comparisons to find a current item and n comparisons to determine if the requested item is not in the current array. This is a hint of an important issue: namely, **what does *average* mean?**

We usually have to indicate the size of the array and would then have to allocate a new array if the number of entries exceeds the array size.

We need some memory management system for dynamic dictionaries. But this is true for any data structure.

Some pros and cons of an ordered list in an array

Note: This is only applicable if the items or the identifiers can be ordered which is usually the case.

Can search for an item in at most $\approx \log_2 n$ comparisons. Doing an asymptotic analysis of the time (and memory) for an algorithm is one of the main aspects in the analysis of an algorithm. Of course, correctness of the algorithm is paramount.

Some pros and cons of an ordered list in an array

Note: This is only applicable if the items or the identifiers can be ordered which is usually the case.

Can search for an item in at most $\approx \log_2 n$ comparisons. Doing an asymptotic analysis of the time (and memory) for an algorithm is one of the main aspects in the analysis of an algorithm. Of course, correctness of the algorithm is paramount.

$\log_2 n = x : 2^x = n$. Note that x will not be an integer unless $n = 2^k$ for some k .

To be precise the worst case number of comparisons is $\lfloor \log_2 n \rfloor + 1$ where the floor function is defined as $\lfloor x \rfloor =$ the largest integer $k \leq x$. **You can verify that for $n = 2^k - 1$, the worst case number of comparisons, is k .**

Ordered lists in an array continued

The differences between $\log n$ and n , can be dramatic (say if a search is within a *loop* of instructions). I will define what we mean by $T(n) = O(f(n))$, $T(n) = o(f(n))$, $T(n) = \Omega(f(n))$ and $T(n) = \Theta(f(n))$. We use this notation to express asymptotic bounds. We will be discuss further the importance of complexity issues and will use this asymptotic notation.

It is more difficult to insert and delete records or modify the identifier of a record even for a fixed size array although updating the content of a record is easy once the item is accessed.

We can easily identify the i^{th} largest or smallest element in a balanced binary search tree..

Tables of some complexity bounding functions

Table 2

Polynomial-Time Algorithms Take Better Advantage of Computation Time

Time Complexity	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
n	0.00001 second	0.00002 second	0.00003 second	0.0000 second	0.00005 second	0.00006 second
n ²	0.0001 second	0.0004 second	0.0009 second	0.0016 second	0.0025 second	0.0036 second
n ³	0.001 second	0.008 second	0.027 second	0.064 second	0.125 second	0.216 second
n ⁵	0.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2 ⁿ	0.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3 ⁿ	0.059 second	58 minutes	6.5 years	3855 centuries	2 × 10 ⁸ centuries	1.3 × 10 ¹³ centuries

Figure: Figure taken from Garey and Johnson “Computers and intractability : a guide to the theory of NP-completeness”. Time in seconds based on an estimate of computers in the late 1970s. What if today computers are 100 times faster. Does this change the “message” in this figure.

A linked list

Introduces the idea of a pointer

A linked list

Introduces the idea of a pointer

I wil show a singly linked list on the board. Can have a doubly linked list.

A linked list

Introduces the idea of a pointer

I wil show a singly linked list on the board. Can have a doubly linked list.

Easy to add items if the list is unordered. If list is ordered then have to follow pointers to see where to insert a new item.

A linked list

Introduces the idea of a pointer

I wil show a singly linked list on the board. Can have a doubly linked list.

Easy to add items if the list is unordered. If list is ordered then have to follow pointers to see where to insert a new item.

May have to traverse the entire list to find an item or determine it is not there.

A balanced binary search tree

Informally, a balanced binary tree with n “nodes” will have depth $\log_2 n$ and hence we can search a balanced binary search tree in at most $\log_2 n$ “edge” traversals and comparisons. Furthermore, with a balanced binary search tree, we can find the smallest (or largest) entry in the search tree in $\log_2 n$ edge traversals and comparisons.

I use the terminology of nodes and edges as a *tree* (in the sense of a search tree) is a special case of a *graph*. Graphs are also referred to as *networks* in many contexts (i.e. a social network, a transportation network, etc.).

The nodes (also called vertices) and edges (also called arcs in some applications) can be undirected or directed. In the latter case, we call a graph with directed edges a *directed graph* and we usually mean an undirected graph if we just say graph.

We will discuss further graph concepts as the term progresses.

A programmer or a programming language could create their own *graph data type*.