CS 234 Course Note

December 16, 2022

1 CS 234: Data Types and Structures

This course note is made by Richard Dong for the Fall 2022 offering of CS 234: Data Types and Structures at the University of Waterloo.

2 Module 1: Users

contract is an interface between **user** and **provider**, as it represents all the shared information that both user and provider need to know

In CS 234, an **abstract data type (ADT)** serves as the contract. - The ADT specifies one or more types of data items and operations on those items. It does not specify how the ADT is to be used, nor how it is to be implemented. - Note: A data type is a data storage format for a programming language, such as a string. Do not confuse the terms data type and abstract data type. - An abstract data type (or, for short, an ADT) - consists of **a type or types of data** and a **set of operations**. (in python term: **class**) - is a specific entity that **stores data** and **supports the set of operations** (in python term: **objects**)

2.1 Key points

Both user and provider agree on the ADT.

The user of the ADT is able to use the operations of the ADT. The user of the ADT does not need to know the details of how the operations work.

In order for the ADT to function, the provider must provide an algorithm, or sequence of steps, to execute each operation as well as an implementation that specifies how the data is stored. - We will consider various data structures, which are ways of storing data in a computer's memory.

A provider implements the ADT operations without knowing how they are used.

2.2 Modularity and data hiding

Modularity is the division of a program into independent, reusable pieces.

Data hiding is the protection of data from other parts of the program.

2.3 Types of data for ADTs

general data (most wide) - operations rely on being able to determine if data items are **equal**, but **not** compare them for ordering or use functions that extract information from the data item.

orderable data - operations rely on determining **equality and/or order** (possibly including $\geq$, $\leq$, $>$, $<$, depending on whether data items are distinct), but not functions that extract information from the data item.

digital data - operations rely on functions that **extract information** from the data items, such as decomposing a string into symbols or applying arithmetic operations to a number. **Comparisons for equality and order** are also allowed

It is important to observe that the same type of data, such as a number, might be treated as general data in one ADT, as orderable data in another ADT, and as digital data in yet another ADT. That said, not all types of data can be treated as orderable or digital data. - **Loosening** requirements on data might widen the range of situations covered. - **Tightening** requirements on data might widen the range of implementations possible.

2.4 Pre- and Post-Conditions

Preconditions: requirement that must be satisfied for an operation to be guaranteed to work. - what is required of the user

Postconditions: a guarantee of the outcome of an operation being executed - if the preconditions are satisfied, then the provider has the obligation to ensure that the postconditions are met

2.5 Choosing an ADT

General and Specific ADTs - A more general ADT supports a larger number of operations, but often at the cost of having **higher costs** for all operations. - A more specific ADT supports a smaller number of operations, sometimes more cheaply.

We use the **Guiding principle**: - Choose an ADT that provides all the operations needed but not too much more.

2.6 Modifying an ADT

Two options: - **Augmenting** a known ADT by adding one or more new operations - The simplest, and primary, way that we will use augmentation is by taking an ADT designed for a simple data item and modify it so that it can handle **compound data**, that is, a data item formed out of several **fields**, each containing data of a particular type. - We consider compound data as distinct from the three major types of data - **Restricting** a known ADT by removing one or more operations - One such circumstance arises when we have **static data**, that is, when the data in the ADT (object) does not change

2.7 Common ADT Operations

2.7.1 Create

Invariably, an ADT (object) will need to exist in order to store data. The two main ways that an ADT (object) will be created is as storing none of the data or all of the data that will be used. On occasion, a create operation might also take as input either data items with which to fill the ADT (object) or information such as size.

2.7.2 Search

In the ADT Multiset, search is based on the value of the data item being sought. Other types of search might be based on the position of an item, where the definition of the term position depends on the particular ADT. It is important to distinguish between position as defined in the ADT and location within an implementation; the latter is not known by the user of the ADT, and hence cannot be specified as part of an input. Other types of search might result in multiple data items, all fitting specified criteria.

2.7.3 Modification

Just like search can be based on position or value, so can addition, deletion, and modification of data items. As noted above, the position is defined in the ADT and may have no correlation with location in an implementation.

2.7.4 Other Operations

Other operations include determining the status of data items or of the ADT, such as determining whether or not any data items are currently being stored.

2.7.5 Planning

Once the user and the provider have agreed on an ADT (class), each can work independently on planning. The user will write an **algorithm that solves the problem**, using the ADT operations. The provider will assess various types of implementations and **algorithms for the operations**, and give the user a choice among them.

We can think of each algorithm as being composed of smaller tasks, roughly divided into two categories: - computation, used by both the user and the provider - done using **pseudocode** and **algorithm sketch**

3 Module 2: Cost Analysis

Goal: Form a function by choosing a representative for each instance size.

3.1 Definitions

Best case: the value of f(n) is the **fastest** running time of the algorithm on *any* instance of size n - Note: The best-case running time of an algorithm is a function that indicates how the smallest representative for each instance size grows as n grows. We are not asking the question "For which instance size is the running time fastest?" since the answer will usually be the smallest instance sizes, telling us nothing about how to compare our algorithm to other algorithms in general. - It should ALWAYS be a function of the INSTANCE SIZE (hence f(n))

Average case: the value of f(n) is the sum over all instances I of size n of the probability of I multiplied by the running time of the algorithm on instance I.

Worst case: the value of f(n) is the **slowest** running time of an algorithm on any instance of size n

We will typically use worst-case running times.

3.2 Catagories

We will be defining catagories using **simple functions**, which retains only the dominant terms. Some common ones include: - $\Theta(1)$: all constant functions - $\Theta(logn)$: all logarithmic functions - $\Theta(n)$: all linear functions - $\Theta(n^3)$: all cubic functions - $\Theta(2^n)$: some exponential functions

3.3 Asymptotic notation (order notation)

f(n) is in $\Theta(g(n))$ if there are real constants c1 > 0 and c2 > 0 and an integer constant $n_0 \geq 1$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for every $n \geq n_0$

We say two functions are asymptotically the same if they are both in $\Theta(f(n))$ for the same function f(n).

3.4 Partial information: Big O and Big Omega

O: f(n) is in $O(g(n))$ is there is a real constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every $n \geq n_0$

Ω: f(n) is in $\Omega(g(n))$ is there is a real constant c > 0 and an integer constant $n_0 \geq 1$ such that $cg(n) \leq f(n)$ for every $n \geq n_0$

Each of O, Ω, Θ can be used to describe any type of function (worst-case, average-case, best-case, or a function that is not a running time).

3.5 Multiple Variables

Keep all the dominant terms, but make sure to treat variables seperately.

3.6 Sum of Functions

To form the sum of functions, keep only the dominant term (or terms, if there are multiple variables) among the categories, - use Θ if all the chosen categories use Θ, and - use O if any one of the chosen categories uses O.

3.7 Product of Functions

To form the product of functions, take the product of the categories, keeping only the dominant term or terms, - use Θ if all the chosen categories use Θ, and - use O if any one of the chosen categories uses O.

Variables must remain variables, and a variable n does not need to appear in the expression of a function on n (as it could not depend on n at all).

3.8 Analyzing parts of algorithms

At times, most often in the role of the user, we may make use of a **placeholder** in our analysis for situations in which the values are not known. For example, if the running time of the helper function is f(n) , we can safely express the running time of the algorithm as $\Theta(f(n))$.

3.9 Branching

The worst-case cost will be the cost of the most expensive branch, including the cost of evaluating all the conditions needed to reach that branch.

For best-case analysis, it is necessary to determine whether or not there exists an input on which the cheapest branch is executed, and if doing so results in the lowest cost for the entire algorithm.

3.10 Looping

The cost of looping will be the cost of all the iterations that are executed. Typically, for worst-case analysis all iterations will be executed. An early exit might be possible for best-case analysis, but you need to ensure that there exists an input on which early exit is possible, and if so, whether the cost of the rest of the algorithm on that input is the lowest possible.

In general, analyzing loops can be quite involved. Fortunately for us, most of the situations in this course are quite simple. - **Common situation 1**: If each iteration of the loop has the same cost, all we need to do is take the product of the number of iterations and the cost of one iteration. - **Common situation 2**: If the cost of an iteration grows linearly with the iteration number, the total cost is quadratic in the number of iterations.

4 Module 3: Provider

4.1 Compound Data

We we store data, a chunk of memory is allocated. - different sizes of chunks may be required for different types of data, such as numbers and strings - Memory can also be allocated for **compound data**, sufficient to store each of the pieces of data of which it is comprised - We can view the portion of the chunk used to store one of the constituent pieces of data as a sequence of contiguous cells in a chunk, or **subchunk**, of the appropriate size

4.2 Arrays

An array is a chunk of memory that can be divided into a sequence of **slots** (subchunks), each of which can store a data item, or element. Each subchunk contains the **same number of cells**. - This is not the same as the Python data type array or the Python data type list.

The *size of the array* is the number of slots. The image shows an array of size 4. Each slot has an index in the range from 0 to one less than the size (so up to 3 in this case).

We use T[i] to denote the contents of slot i in array T

Conventions in drawing of memory - A grouping of subchunks may be depicted without showing the chunk as well. - A blank space may represent either an empty chunk or subchunk or a generic, unspecified data item, depending on the context. - A question mark is used to represent a chunk or subchunk with an unknown value, such as may result from allocation without initialization. - Initialization or other writing is needed before reading.

Since each slot can be located by using the address of the chunk and the size of slot, reading or writing a slot can be accomplished in **constant time**, regardless of the index.

The array is the main data structure used in a **contiguous memory implementation** that we will consider in the course, so named due to the fact that the entire array is stored in one chunk. - An implementation does not need to consists of a single array. An implementation might use more than one array, and/or may include one or more variables storing information such as a single number or string.

Definition of the Implementation - The data structures (and possible other ADTs) used in the implementation - The meanings of values of variables, if any - Rules governing how data is arranged in each data structure, including ordering of data items, if any

Once an implementation has been chosen, there may be many possible algorithms that implement the ADT operations. It is important that all algorithms preserve the definition of the implementation.

Searches in Arrays

Although we can easily access slots in any order we wish, it is easiest to write a loop that considers each slot index, typically in increasing order. A search of this type is known as **linear search**. We can customize the linear search for a particular algorithm by specifying when it stops, such as when the data item being sought has been found or at a particular index.

In discussing search algorithms of any type (not limited to linear search), we distinguish between a **successful search**, in which a data item is found, and an **unsuccessful search**, in which it is determined that the data item is absent. We will find that at times the costs of successful search and unsuccessful search can differ, especially when considering best-case or average-case running times.

Costs

Allocation a chunck of any size, finding a slot, reading a slot, and writing a slot all cost $\Theta(1)$, yet the **cost of initialzing** contiguous memory depends on the **number of slots**.

Unless stated otherwise, you can assume that we have enough space for all data item additions in arrays used in future implementations. You should not make any assumptions about whether or not memory for an array has been initialized, so be careful to watch for that detail.

4.3 Linked Memory

In a **linked implementation**, small chunks of memory can be requested as they are needed. The reason for the name of the implementation is that each chunk, or **linked node**, will consist of a small number of subchunks, storing not only the data item but also one or more *addresses of other linked nodes*.

Instead of being interpreted as a number, an address is viewed as a type of data known as a **pointer** or link, as it is a way of pointing from one linked node to another, or, equivalently, linking one linked node to another. A linked node is said to point (or link) to another linked node if the first linked node contains a pointer to the second linked node. If a node does not point to any other node, it stores the empty pointer, known as the **null pointer**.

In our diagrams, pointers will be represented by arrows, and empty or null pointers by diagonal lines.

4.4 Linked Lists

It consists of a sequence of zero or more **linked nodes** (each allocated as a chunk of memory) as well as a variable **Head** storing a pointer to the first node in the sequence (if any). The variable Head stores a null pointer if the linked list is empty (that is, it contains no linked nodes). - we'll refer to the **position** of a linked node in a linked list, where the linked node in the first position is the linked node to which **Head points** (if one exists)

More generally, the linked node to which the linked node l points is the **successor** of l, and the linked node that points to l is the **predecessor** of l.

5 Module 4: Position

What is missing from both the ADT Multiset and the ADT Set is the notion of a **position** of a data item. When we referred to the "most recently added" data item, we were implicitly referring to position. We can think of the data as being placed in a pile, with the most recent data item at the top of the pile.

5.1 Terminologies

Push: add an item to the top of the stack

Pop: return the top data item and delete it

5.2 Circular Array

It still has variables for the front and the back, but now we allow the index for the back of the queue to be smaller than the index for the front of the queue

5.3 Seperation of ADTs and Implementations

- An ADT is not a data structure.
- An ADT that behaves like contiguous memory does not need to be implemented using contiguous memory.
- Linked and contiguous implementations needn't be obvious ones.

5.4 Choices of Implementations

- A data structure can consist of multiple arrays.
- An array can store more than one piece of information in a slot.
- A linked list can store more than one piece of information in a linked node.

6 Module 5: Trees

6.1 Terminologies

A tree is formed of **nodes** connected by **edges**. This is not the same as a node in a linked list; at times we'll write **tree node** to distinguish it from a linked node.

The two nodes connected by an edge are the **endpoints** of the edge.

In a **rooted tree**, one node is designated as the **root**, and contrary to what you might expect from trees you've seen in real life, the root appears at the *top* of the drawing.

When two nodes are connected by an edge, the node closer to the root is the **parent** and the node farther from the root is the **child**.

Nodes with the same parent are **siblings**.

Continuing with the family tree idea, the **ancestors** of a node are its parent, its parent's parent, and so on up to the root

The **descendants** of a node are its children, children's children, and so on down to the leaves

A **leaf** is a node with no children, and any node that is not a leaf is an **internal node**.

A tree is **unordered** if there is no order specified on the children of a node, and **ordered** otherwise. - We can also define a special type of ordered tree. A **binary tree** is a tree in which each parent has at most two children and each child is specified as either a **left child** or a **right child**. - Notice that this means that if there is a single child, it must be identified as either a left child or a right child. It can't just be a generic child.

A node and all its descendants form the **subtree** rooted at that node. - In a binary tree, the subtree rooted at the left child of a node is the **left subtree** of the node and the subtree rooted at the right child of the node is the **right subtree** of the node.

A **path** is defined as a sequence of nodes such that there is an edge between each consecutive pair of nodes in the sequence. The **length** of a path is one less than the number of nodes in the path, or, equivalently, the number of edges in the path.

Unless stated otherwise, we'll assume that each path is **simple**; each node appears at most once in the sequence

If we're measuring lengths of paths from the root, we use the term **depth**, where the depth of a node is the length of the path between that node and the root. All nodes of the same depth are on the same **level**.

To measure the length of a path from a leaf to a node, we use the term **height of a node** to refer to the maximum length of a path to a leaf in the subtree rooted at that node. As you can see in the image, every leaf has height 0, and the height of a node is one greater than the maximum height of any of its children. - As you can see in the image, every leaf has height 0, and the height of a node is one greater than the maximum height of any of its children. - The largest height of any node is the **height of the root**, which is also known as the height of the tree. The height of the root of the tree will be equal to the height of the tree.

In a **perfect binary tree**, each node has zero or two children and all leaves are at the same depth. - For any $k \geq 0$, there exists a perfect binary tree with $2^k - 1$ nodes

In a **complete** binary tree, every level, except possibly the last, is completely filled, and all nodes on the last level are as far to the left as possible.

6.2 Recursive Definition of a Tree

A tree is either: - empty or - a node with ezro or more subtrees, where each subtree is a tree

6.3 Implementations

6.3.1 Linked Binary Tree

Variable Root storing the null pointer or a pointer to the root node; each linked node stores a data item, a pointer Parent to the parent, if any, and otherwise the null pointer, a pointer Left to the left child, if any, and otherwise the null pointer, and a pointer Right to the right child, if any, and otherwise the null pointer - If Root stores the null pointer, the ADT is empty. - Each linked node stores a data item.

6.3.2 Contiguous Implementation

For the node at index p: - The index of its left child, if any, is 2p+1 - The index of its right child, if any, is 2p+2 - The index of its parent, if any, is $\lfloor (p-1)/2 \rfloor$

Since we would like to be able to easily find the parent, left child, or right child of a node, we'll sacrifice space to ensure that the operations can be easily implemented.

6.3.3 Tree Traversal

- In a **level order traversal**, nodes appear in nondecreasing order of **depth**.
- In a **postorder traversal**, each node appears after its children.
- In a **preorder traversal**, each node appears before its children.
- In an **inorder traversal** (defined only for a binary tree), for each node, all nodes in the left subtree come before the node and all nodes in the right subtree come after the node.
  - There is exactly one inorder traversal of a given binary tree.

In a linked implementation, we can **thread** nodes together by adding an extra pointer from a node to the next node in whichever traversal order we wish. We could also add two pointers, to the successor and predecessor, to form a doubly-linked list of all the nodes.

7 Module 6: Graphs

7.1 Terminologies

Just like in a tree, we are using **edges** to connect pairs of data items, allowing us to store data items in positions with respect to other data items. - However, in a graph, there is nothing equivalent to a root, which means that we lose the concepts of root, parent, child, and so on. - Another big difference is that although in a tree there is exactly one path between any two data items, in a graph there can be zero, one, or more than one path between a pair of data items.

Whereas trees have nodes, graphs have **vertices**. It's really the same idea, and sometimes the same terms are used, but I'm using different terms to be able to keep them clear. We can now formally define a graph: a graph G is a set V(G) of vertices and a set E(G) of edges - The word vertices is the plural form of the word; the singular is vertex

We don't yet have any notion of an edge being directed from one vertex to another, as this is an **undirected graph**. - We can express each edge as the set of the two vertices it connects (such as {u,v} meaning the edge connecting u and v)

A graph is **simple** if there can be at most one edge between any pair of vertices, and for any edge {u,v}, $u \neq v$.

Two vertices u and v are **adjacent** if {u,v} belongs to E(G).

The vertices u and v are the **endpoints** of the edge {u,v}.

The edge {u,v} is **incident on** the vertices u and v(and vice versa).

Two edges are **incident** if they share an endpoint.

We might wish to know about all the vertices adjacent to a particular vertex; fittingly enough, we call that a **neighbourhood** and each vertex in it a **neighbour**. - The number of neighbours is the **degree** of a vertex, which can also be described in terms of the number of incident edges. - If a vertex has degree zero, it is **isolated**.

This path (from u to v) is a **simple path** if no vertex is repeated.

If there is also an edge linking the last vertex to the first vertex in the sequence, we have a **cycle**. - If a graph has no cycles, that is, it is **acyclic**.

A graph is **connected** if there is a path between any pair of vertices. - A connected graph with the maximum number of edges is called a **complete graph** (ie: there is an edge between any pair of vertices)

We will use the term **directed edge** or **arc**, typically drawn as an arrow to show the direction. - A graph in which edges are directed edges is called a **directed graph**

An arc is written as an **ordered pair** of endpoints (u,v), where u is the **tail** and v is the **head**

We can have two arcs with the same endpoints in a simple directed graph, as long as the two arcs point in two different directions.

if c -> a -> b, then a has one **in-neighbour** (c) and one **out-neighbour**. It has **indegree** of one and **outdegree** of one.

There can be mutliple **directed paths** from vertex a to vertex b.

7.2 Running Times

For m is the number of edges and n is the number of vertices

$$m \, \epsilon \, O(n^2)$$

for any graph. For a conncted graph:

$$m \, \epsilon \, \Omega(n^2)$$

The degree of any vertex is in $O(min\{m, n\})$

The sum of degrees of all vertices is 2m.

7.3 Implementation

In the **edge list** implementation, the edges are stored as a linked list in which each linked node contains the vertex IDs of the endpoints of an edge and a pointer to the next edge in the list. - Easy accommodates arbitrary vertex IDs if no vertex attributes

Adjacency Matrix: - The entries (One, Two) and (Two, One) store information about the edge with endpoints One and Two in an ADT Grid. - Fast ARE_ADJACENT method

Adjacency list - we have an array with one slot for each vertex, with a pointer to a linked list consisting of all the neighbours of that vertex. Below is an example of what this would look like. - Fast NEIGHBOURS method

7.4 Dept-First Search (DFS) and ## Breadth-First Search (BFS)

7.4.1 DFS

Search all the neighbours of a vertex. When its done, colour is black and go back to the previous vertex. Repeat untill all reachable vertices have been visited (ie: there is no neighbours left.

7.4.2 BFS

Searches broadly by considering neighbours, then neighbours of neighbours, and so on.

7.4.3 Running Time for Both Search

The time will depend on the implementation - Edge List: $\Theta(nm)$ - Adjacency Matrix: $\Theta(n^2)$ - Adjacency List: $\Theta(n+m)$

8 Module 7: Dictonary

8.1 ADT Dictionary

The ADT Dictionary supports (key, element) pairs accessed by key, where - keys are **distinct** but not necessarily orderable, and - elements are general data.

8.2 Searching

When we have general data, in the worst case we need to examine each data item in both successful search and unsuccessful search, typically using linear search.

To allow a richer set of possible implementations and algorithms, in the rest of the module we will consider the special case in which keys are orderable. We expand the definition of an **ordered array** and a **ordered linked list** to refer to the ordering of keys, allowing us to define additional implementations.

8.3 Sorting

Sorting is often used when it is not needed. When you are tempted to sort, first figure out whether sorting is necessary to solve your problem.

A sorting algorithm is **stable** if equal-valued items are in the same order before and after sorting.

To add a new item: - For ordered array: - Sort data items in nondecreasing order by key, and then insert them in consecutive slots, starting at slot 0. - For ordered linked list: - Sort data items in nonincreasing order by key, and then insert each data item, in order, at the beginning of the list.

8.4 Binary Search Tree (BST)

BST satisfies the binary search tree property, that is, for every node Node in the tree, for Key the key stored in the node: - The key values stored in the **left** subtree of Node are **less** than Key. - The key values stored in the **right** subtree of Node are **greater** than Key.

8.4.1 Look_Up (D, Key) for BST

- If the key being sought is equal to the key stored in the node, we stop and return the element in the node.
- If the key being sought is **smaller** than the key stored in the node, we continue the search at the left child of the node, if any, and otherwise stop and return False.
- If the key being sought is **larger** than the key stored in the node, we continue the search at the right child of the node, if any, and otherwise stop and return False.

8.4.2 Delete(D, Key)

Case 1: - Searches for the node Node storing Key. - If Node has no children, deletes Node.

Case 2 - Searches for the node Node storing Key. - If Node has **only one child** Child: - Determines whether Node is a left or right child. - If Node is a **left** child, makes the **child of Node** into the **left child of the parent** of Node. - If Node is a **right** child, makes the **child of Node** into the **right child of the parent** of Node.

Case 3 - Searches for the node Node storing Key. - If Node has **two children**: - Finds the **inorder successor** Succ of Node. - Swaps Node's (key, element) pair with Succ's (key, element) pair. - Deletes Succ using Case 1 or 2. - The worst-case cost of finding the inorder successor or inorder predecessor of a node is linear in the height of the tree. - Our algorithm depends on the use of an inorder successor or predecessor as well as the fact that the **inorder successor or predecessor has at most one child**, so that after Case 3 we can use Case 1 or Case 2.

8.5 AVL Tree

A perfect tree has a high that is $\Theta(log(n))$, where n is the number of nodes.

8.5.1 Tradeoff in degree of organization of data

- Organizing the data **more** might lower the cost of **search**.
- Organizing the data **less** might lower the cost of **modification**.
- Goal:
  - Maintain $\Theta(log(n))$ height
  - Implement $\Theta(log(n))$ for the worst case time for Look_Up, Add, and Delete

8.5.2 Balance

We'll say that a node is **balanced** if the difference in heights of the left and right subtrees is at most 1, and **imbalanced** otherwise. A tree satisfies the **height-balance property** if every node is balanced.

An AVL Tree is a height-balanced tree and has a height of $\Theta(log(n))$.

8.5.3 Rebalancing

Implementing an Add or a Delete operation like in a BST will preserve the binary seach tree property but might violate the height-balance property.

As part of each operation, we will **rebalance** the tree. To do so, we will first identify the **pivot node**, the lowest imbalanced node in the tree after an insertion or deletion.

A **rotation** on the pivot node is a rearrangement of subtrees that rebalances the tree without violating binary search order. - For Add, rotation only requires for the substree - For Delete, rotation requires for the entire tree by tracing the path to the root

8.5.4 Rotations

There are three points in time: - Before modification - After modification - After rotation

Goal - Restore the height-balance property - Preserve the binary search tree property - Change a constant number of pointers

Observations - After modification, the only nodes that might be imbalanced are the pivot node and its ancestors. - After modification, the difference in heights of the left and right subtrees of the pivot node will be exactly 2.

Cases - Case 1 - both higher pivot subtree and higher subtree are left - Hight of subtree for Case 1 - After modification: h+3 - After rotation: h+2 - Case 2 - higher pivot subtree is left and higher subtree is right - Hight of subtree for Case 2 - After modification: h+3 - After rotation: h+2 - Case 3 - higher pivot subtree is left and subtrees have same height - Hight of subtree for Case 2 - After modification: h+3 - After rotation: h+3

8.6 (2,3) Tree

Def - Each internal node has either one key and two children or two keys and three children. - All leaves are at the same depth and have one or two keys. - Keys in the left subtree are smaller than the first or only key, in the middle subtree (if any) are between the first and second keys, and in the right subtree are greater than the second or only key.

Any (2,3) tree sotring n data items has height $\Theta(log(n))$

8.6.1 Overflow

When we add a key, it results in three keys in one node - We **split** the node by moving the middle one to the parent - If splitting leads to overflow in the parent, then the parent may need to be split as well. - Splitting can propagate up to the root.

Algorithm for split - A node with three keys becomes two nodes, one with the smallest key and one with the largest key. - The middle key is sent up to the parent. - If the node being split was the root, the tree now has a new root.

8.6.2 Underflow

- If there are too few values in a node, then underflow has occurred.
- The fuse operation fuses two nodes into one and rearranges values as needed.

9 Module 8: Priority Queues

We want an ADT that can look up the minimum quickly

9.1 Comparing implementations

9.1.1 Ordered

- Delete_Min is **fast** as the minimum can be found easily
- Add_Min is **slow** as mintaining sorted order is costly

9.1.2 Unordered

- Delete_Min is **slow** as the minimum cannot be found easily
- Add_Min is **fast** as no order needs to be maintained

9.1.3 Middle Ground

- Organizing the data more might lower the cost of search.
- Organizing the data less might lower the cost of modification.
- Goal:
  - No operations as slow as $\Theta(n)$
  - $\Theta(1)$ worst-case cost for Look_Up_Min

9.2 Heap

A binary tree satisfies the **heap-order property** if for each node, the key stored at the node is no greater than that stored in either child.

Putting together the ideas we've developed so far, we'll make use of a **heap**, which is a **complete binary tree** that satisfies the heap-order property.

The worst case running time to convert a complete binary tree to satisfy the heap-order property is $\Theta(n)$

We can create a sorting algorithm called **heapsort**: - Build a heap by using Heapify - Extract pairs in order using Delete_Min repeadedly

10 Module 9: Improvements

Recall: - Best searching (binary search): $\Theta(log(n))$ - Best sorting: $\Theta(nlog(n))$ - Linear Search: $\Theta(n)$

All those have the same average cost and worst case cost.

10.1 Interpolation Search

If we tweak binary search such that at each phase for an array A: - consider positions Low through High - $Mid = Low + \frac{Data - A[Low]}{A[High] - A[Low]} \times (High - Low)$ - At each phase, Data is compared to A[mid] - If Data is bigger, update Low to Mid+1, otherwise update High to Mid-1

Worst case running time is $\Theta(n)$

Average case running time is $O(log(log(n)))$

10.2 Heuristics

Static Data Structure: in addition to not changing the data (like the case with static data), we also do not change how the data is arranged.

We now use heuristic instead of an algorithm - A method that may not guarantee bounds on running time or correctness - to ensure the smallest worst-case cost, put the items in nonincreasing order of probability. Doing so results in higher probabilities being multiplied by smaller costs. - But what if we do not know the probability of selecting future events? - Use **self-organizing heuristic**: we use information about past accesses to rearrange data items.

Move to Front

if a data item was accessed, it is likely to be accessed again - After a successful search, the data item that was sought is moved to the front of the list (if it is not already there)

Transpose

After successful search, the data item that is found is exchanged with the data item preceding it, if any. - Using Transpose, the most-used element moves slowly towards the front, and the least-used element moves slowly towards the back.

10.3 Hashing

The term **hashing** refers to a general way of mapping keys to integers in the range, or equivalently, of distributing keys into **buckets**.

10.3.1 Hashing Method

- Each specifies a **hash function** f that maps keys (or data items) into buckets.
  - That is, each key is mapped to a value in the range from 0 to N - 1, where N is the number of buckets
- Each specifies a method for **collision resolution**
  - a collision occurs when two different keys are mapped by f to the same bucket

10.3.2 Hash Functions

Goals: - Quick to compute - Spreads keys evenly among buckets

10.3.3 Conflict Resolution

- **Separate chaining**: Store all data items k with f(k) = i in bucket i
- **Open addressing**: Store at most one data item in each bucket

By ensuring that the computation of the hash function can be executed in constant time and the examination of each data item or location can be executed in constant time, we can measure the cost of a search as the number of **probes** needed to find a data item, where each probe is the examination of a data item or location. - The average number of probes needed for a search is compared to the **load factor**, which is the ratio of the number of data items and number of buckets.

Method 1: Separate Chaining

All items that map to a bucket are stored in that bucket.

Method 2: Open Addressing

Characteristics: - Each bucket is associated with a slot in an array. - Store at most **one item** in each bucket. - A data item k might not be stored in the bucket f(k) - Operations search through slots in a particular order for a particular data item

The **probe sequence** for a data item is the order in which buckets are searched to find the data item or to find a free bucket in which to store the data item. - **Linear Probing** - When we are adding a data item with key k, if slot f(k) is full, we look at the next one, then the one after that, and so on, stopping when we find an empty slot and place it in the slot - **Clustering**: the phenomenon in which once a collision occurs, more collisions tend to pile up in the same place, like a snowball growing bigger and bigger. - **Quadratic Probing** - instead of having the size of the "step" between consecutive slots in each probe sequence be the same, we have different "step" sizes. - In quadratic probing, it will be 1^2, 2^2, and so on - Although quadratic probing will perform better than linear probing in some cases, notice that if the hash function maps two keys to the same bucket, then they will each have exactly the same probe sequence. - **Double Hashing** - We use a second hash function, g(k), to determine by far how to "jump" to get to the next bucket in the probe sequence - We do have to be a little careful, since we want to make sure that our probe sequence includes all buckets

10.4 Sorting

10.4.1 Bucket Sort

Steps: - Initializes each entry in an array of length r to empty - For each data item (i, value), use add_to_bucket(Bi, value) - For each bucket Bi, in order, concatenates the results of Return_Contents(Bi)

Worst case cost is $\Theta(n + r)$, where r is the number of items in the bucket. We want to be able to compete with $\Theta(nlog(n))$ even for large r

10.4.2 Radix Sort

Steps: - Divides each value into **chunks** of bits - For each chunk from rightmost to leftmost, uses a **stable bucket sort** of all values based on the current chunk

Worst case cost is $O(ln)$, where l is the number of chunks or the number of passes.

11 Module 10: Extensions

11.1 Skip List

Features: - Combines strong points of linked (easy to modify) and contiguous (binary search) - Linked nodes for different data items of different "heights" - Linked nodes at the same level are linked in a doubly-linked list - Extra nodes at left and right of each level for technical purposes

11.2 Multi-Dimensional Data

Searches: - Range queries: Find all data items in a certain specified range - We can use an inorder traversal - Nearest-neighbour queries: Find the data item closest to a specific query point - Partial

key search: Find a data item specified by only certain dimensions

Range Tree for range search - each node stores the maximum value stored in its left subtree. - Would have: - inside nodes: they and their descendants are all in the range. - outside nodes: they and their descendants are all outside the range. - boundary nodes, which have descendants both in and out of the range.

Quadtree for 2D data - Represent a square region as a tree - Root represents entire region - Four children for four quadrants - Leaves store points

k-d tree for less evenly-spaced data - Represent a region as a binary tree - Nodes represent points - Splits depend on distributions of points in a region

11.3 Special Cases

11.3.1 Trie

An ordered tree in which some nodes store strings, some nodes are used as part of the search, and some nodes serve both roles.

To reduce the number of extra nodes, we can instead use a **compressed trie**.

11.3.2 Suffix Tree

A suffix tree stores all the suffixes of a string, which means that the search can take place simultaneously at all starting points.

11.3.3 New Criteria for Data Structures

Kinetic data structures capture the idea of motion.

Persistent data structures allow multiple versions to be stored simultaneously, so that you can handle changes to data over time.

Succinct data structures, which use as little space as possible.

11.3.4 New methods of analysis

Amortized analysis determines the worst-case cost of a sequence of operations.

11.3.5 Memory

The memory hierarchy consists of different types of memory, where the size of each type increases as the access speed decreases.

It is often enough just to consider two levels of memory, where main memory is cheaper and in short supply and external memory (such as external disks or the cloud) is more expensive but more abundant. Here “cheap” and “expensive” refer largely to the costs of accesses to memory.

11.3.6 Impact on Data Structure Design

In designing data structures, we then need to take into account the cost of moving data from external memory, where it is stored, to main memory, where it is processed. - Not all data can be in main memory at once, so data needs to be moved in and out as needed.

Rather than moving a single data item at a time, a chunk of memory, known as a **page**, is moved at once. - The key idea is that although we can ignore the memory hierarchy when dealing with a small amount of data. - For large amounts of data, page size matters, as the cost of moving memory around might be what dominates the cost of an algorithm.

There are various **paging algorithms** used to determine which page(s) to move out when a new page is moved in.

In a **B-tree** of order d, d data items fit on a page. - As another variant, you can put all the data in the leaves, use internal nodes for search only, and thread (add pointers to link) the leaves together, forming a **B+-tree**.