1 Abstract Data Types (ADTs)

**Definition.** An abstract data type is a set of mathematical objects and a set of operations that can be performed on these objects.

**Examples**

1. **ADT: INTEGERS**
   - **objects:** integers
   - **operations:**
     - ADD\((x, y)\): add \(x\) and \(y\)
     - SUBTRACT\((x, y)\): subtract \(y\) from \(x\)
     - MULTIPLY \((x, y)\): multiply \(x\) and \(y\)
     - QUOTIENT \((x, y)\): divide \(x\) by \(y\)
     - REMAINDER \((x, y)\): take the remainder of \(x\) when dividing by \(y\)

2. **ADT: STACK**
   - **objects:** elements, stack
   - **operations:**
     - PUSH\((S, x)\): adds the element \(x\) to the end of the list \(S\)
     - POP\((S)\): deletes the last element of the nonempty list \(S\) and returns it
     - EMPTY\((S)\): returns true if \(S\) is empty, false otherwise

2 Data Structures

**Definition.** A data structure is an implementation of an ADT. It consists of a way of representing the objects and algorithms for performing the operations.

**Examples**

1. **ADT: INTEGERS**
   - **objects:** An integer is stored as one word of memory on most machines.
   - **operations:** ADD \((x, y)\) is often implemented in the Arithmetic Logic Unit (ALU) by a circuit algorithm such as “ripple-carry” or “look-ahead.”
2. ADT: STACK
   
   **objects:** A stack could be implemented by a singly-linked list or by an array with a counter to keep track of the “top.”

   **Exercise:** Can you think of any advantages or disadvantages for implementing the STACK ADT as an array versus implementing it as a singly-linked list?

   **operations:**
   
   **Exercise:** How would you implement PUSH, POP and EMPTY in each of these implementations?

ADTs describe what the data is and what you can do with it, while data structures describe how the data is stored and how the operations are performed. Why should we have ADTs in addition to data structures?

- important for specification
- provides modularity
  - usage depends only on the definition, not on the implementation
  - implementation of the ADT can be changed (corrected or improved) without changing the rest of the program
- reusability
  - an abstract data type can be implemented once, and used in lots of different programs

The best data structure for an algorithm usually depends on the application.

3 Analyzing Data Structures and Algorithms

The complexity of an algorithm is the amount of resources it uses expressed as a function of the size of the input. We can use this information to compare different algorithms or to decide whether we have sufficient computing resources to use a certain algorithm.

**Types of resources:** Running time, space (memory), number of logic gates (in a circuit), area (in a VLSI) chip, messages or bits communicated (in a network)

For this course, the definition of input size will depend on what types of objects we are operating on:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplying Integers</td>
<td>Total Number of Bits Needed to Represent the Integers</td>
</tr>
<tr>
<td>Sorting a List</td>
<td>Number of Elements in the List</td>
</tr>
<tr>
<td>Graph Algorithms</td>
<td>Vertices and Edges</td>
</tr>
</tbody>
</table>

The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed (for example, number of comparisons). This also depends on the problem. We
want the notion of “step” to be machine independent, so that we don’t have to analyze algorithms individually for different machines.

How do we measure the running time of an algorithm in terms of input size when there may be many possible inputs of the same size? We’ll consider three possibilities:

### 3.1 Worst case complexity

**Definition.** For an algorithm $A$, let $t(x)$ be the number of steps $A$ takes on input $x$. Then, the worst case time complexity of $A$ on input of size $n$ is

$$T_{wc}(n) \doteq \max_{|x|=n}\{t(x)\}.$$  

In other words, over all inputs of size $n$, $T_{wc}(n)$ is defined as the running time of the algorithm for the slowest input.

**Example:** Let $A$ be the following algorithm for searching a list $L$ for an element with key equal to the integer $k$:

```
ListSearch (List L, Integer k)
  Element z = head(L);
  while (z != null) and (key(z) != k) do
    z = next(L, z);
  return z;
```

We have several options for what we should count as a “step”. We could count every atomic operation (i.e. assignments, returns and comparisons) or we could count only each comparison. Since we are really interested in the number of comparisons and the total number of atomic operations is within a constant factor, it is reasonable to count only the number of comparisons.

Notice that in each iteration of the loop, $A$ does 2 comparisons. If we get to the end of the list $A$ does a final comparison and finds that $z$ is equal to $null$ (we assume that the “and” checks the first comparison and then the second only if the first was true).

Then, let $n$ be the length of $L$ and let $t(L, k)$ be the number of comparisons performed by ListSearch on input $(L, k)$. Then,

$$t(L, k) = \begin{cases} 
  2i & \text{ for } k \text{ the } i\text{th element of } L \\
  2n + 1 & \text{ if } k \text{ is not in } L 
\end{cases}$$

So clearly $T_{wc}(n) = 2n + 1$. This can be written in asymptotic notation as $\Theta(n)$.

### 3.2 Best case complexity

**Definition.** For an algorithm $A$, let $t(x)$ be the number of steps $A$ takes on input $x$. Then, the best case time complexity of $A$ on input of size $n$ is

$$T_{bc}(n) \doteq \min_{|x|=n}\{t(x)\}.$$
**Example:** We use ListSearch as algorithm $A$ again.

We know that $T_{bc}(0) = 1$ since there is only one list of length 0 and for any value $k$, $A$ will make exactly one comparison when $L$ is empty.

This shows that $T_{bc}(0) = 1$ but does not give any insight into $T_{bc}(n)$ for $n \geq 1$. For $n \geq 1$, let $L = 1 \rightarrow 2 \rightarrow \ldots \rightarrow n$ and let $k = 1$. Exactly two comparisons will be made for this instance of $L$ and $k$. Therefore, $T_{bc}(n) \leq 2$. This is an upper bound on the *best case time complexity* of ListSearch for $n \geq 1$.

If $n \geq 1$, then $\text{head}[L] \neq \text{null}$. Therefore, the first comparison evaluates to true and a second comparison is performed. Therefore, $T_{bc}(n) \geq 2$. This is a lower bound on the *best case time complexity* of ListSearch for $n \geq 1$.

Therefore, $T_{bc}(n) = \begin{cases} 1 & \text{for } n = 0 \\ 2 & \text{for } n \geq 1 \end{cases}$

It should be noted that best case complexity often does not reveal useful information about a problem and we will often ignore it in this course.

### 3.3 Average case complexity

Let $A$ be an algorithm. Consider the sample space $S_n$ of all inputs of size $n$ and fix a probability distribution. Usually, we choose the probability distribution to be a uniform distribution (i.e. every input is equally likely).

Recall that a random variable is a function maps from elements in a probability space to $\mathbb{N}$. Also, recall that the expected value of a random variable $V : S \rightarrow \mathbb{R}$ is $E[V] = \sum_{x \in S} V(x) \cdot \Pr(x)$.

**Definition.** Let $t_n : S_n \rightarrow \mathbb{N}$ be the random variable such that $t_n(x)$ is the number of steps taken by algorithm $A$ on input $x$.

Then $E[t_n]$ is the expected number of steps taken by algorithm $A$ on inputs of size $n$. The *average case time complexity* of $A$ on inputs of size $n$ is defined as

$$T_{avg}(n) \overset{d}{=} E[t_n].$$

The following three steps should be performed before doing any *average case time complexity* analysis:

1. Define the sample space
2. Define the probability distribution function
3. Define any necessary random variables

**Example:** Again, we use ListSearch as algorithm $A$.

It is sufficient when analyzing this algorithm to assume the list $L$ is the list $1 \rightarrow 2 \rightarrow \ldots \rightarrow n$ and $k \in \{0, \ldots, n\}$. More precisely, consider any input $(L, k)$. Let $L' = 1 \rightarrow 2 \rightarrow \cdots \rightarrow n$. If $k$ is the $i$th element of the list $L$, let $k' = i$; if $k$ is not in the list $L$, let $k' = 0$. Since the algorithm only performs equality tests between $k$ and elements of $L$, the algorithm will have the same behavior on $(L', k')$ as it did on $(L, k)$. We use this simplified form so that the sample space of inputs, $S_n$, will be finite and therefore simpler to handle.
1. Sample Space: \((1 \rightarrow 2 \rightarrow \ldots \rightarrow n, k \in \{0, \ldots, n\})\)

2. We will assume a uniform distribution

3. Similarly to before,
   \[ t_n(L, k) = \begin{cases} 
   2k & \text{for } k \neq 0 \\
   2n + 1 & \text{if } k = 0 
   \end{cases} \]

   Note that the assumption of a uniform distribution may not be a reasonable one. If the application will often search for elements \(k\) which are not in the list \(L\), then the probability of input \((L, 0)\) would need to be higher than the other inputs.

   Then,
   \[
   T_{av}(n) = E[t_n] = \sum_{k=0}^{n} \Pr(L, k)t_n(L, k) \\
   = \frac{1}{n+1}(2n+1) + \sum_{k=1}^{n} \Pr(L, k)t_n(L, k) \\
   = \frac{2n+1}{n+1} + \frac{1}{n+1} \sum_{k=1}^{n} 2k \\
   = \frac{2n+1}{n+1} + n 
   \]

   Notice that \(1 \leq \frac{2n+1}{n+1} < 2\), so \(n+1 \leq T_{av}(n) < n+2\). This value is somewhat smaller (as expected) than \(T_{wc}(n)\). Asymptotically, however, it is also \(\Theta(n)\), which is the same as \(T_{wc}(n)\). For some algorithms, \(T_{wc}\) and \(T_{av}\) are different even when analyzed asymptotically, as we shall see later in the course.

3.4 Comparison

   We now have three methods for analyzing the time complexity of an algorithm:

   \[
   \begin{align*}
   \text{Worst Case} & \quad T_{wc}(n) = \max_{|x|=n} \{t(x)\} \\
   \text{Average Case} & \quad T_{av}(n) = E[t(x)| |x|=n] \\
   \text{Best Case} & \quad T_{bc}(n) = \min_{|x|=n} \{t(x)\} 
   \end{align*}
   \]

   Then, from the definition of expectation,
   \[
   T_{bc} \leq T_{av} \leq T_{wc} 
   \]

   Each of these three measures can be useful depending on the algorithm and the application. Some algorithms have large \(T_{wc}\) but small \(T_{av}\) while for other algorithms \(T_{wc}\) and \(T_{av}\) are equal.
3.5 Upper and Lower Bounds

Recall that there is an important distinction between proving upper bounds and proving lower bounds on an algorithm’s worst case running time.

An upper bound is usually expressed using Big-O notation. To prove an upper bound of \( g(n) \) on the worst case running time \( T_{wc}(n) \) of an algorithm means to prove that \( T_{wc}(n) \) is \( O(g(n)) \). This is roughly equivalent to proving that

\[
T_{wc}(n) = \max_{|x|=n} t(x) \leq g(n)
\]

How can we prove that the maximum of a set of values is no more than \( g(n) \)? The easiest way is to prove that every member of the set is no more than \( g(n) \).

In other words, to prove an upper bound on the worst case running time of an algorithm, we must argue that the algorithm takes no more than that much time on every input of the right size. In particular, you cannot prove an upper bound if you only argue about one input, unless you also prove that this is input really is the worse in which case you’re back to proving something for every input.

A lower bound is usually expressed using Big-\( \Omega \) notation. To prove a lower bound of \( f(n) \) on the worst case running time \( T_{wc}(n) \) of an algorithm means to prove that \( T_{wc}(n) \) is \( \Omega(f(n)) \). This is roughly equivalent to proving that

\[
T_{wc}(n) = \max_{|x|=n} t(x) \geq f(n)
\]

How can we prove that the maximum of a set of values is at least \( f(n) \)? The easiest way is to find one element of the set which is at least \( f(n) \).

In other words, to prove a lower bound on the worst case running time of an algorithm, we only have to exhibit one input for which the algorithm takes at least that much time.
4 Dictionaries

A dictionary is an important abstract data type (ADT). It represents the following object and operations:

**ADT: DICTIONARY**

*objects:* Sets of elements $x$ such that each $x$ has a value $key(x)$ such that $key(x)$ comes from a *totally ordered* universe

Note: Totally ordered just means that for any two keys $a$ and $b$, either $a > b$, $a < b$, or $a = b$.

*operations:*

- **ISEMPTY(Set $S$):** check whether set $S$ is empty or not
- **SEARCH(Set $S$, Key $k$):** return some $x$ in $S$ such that $key(x) = k$ or null if no such $x$ exists
- **INSERT(Set $S$, Element $x$):** insert $x$ into $S$
- **DELETE(Set $S$, Element $x$):** remove $x$ from $S$

There are many possible data structures that could implement a dictionary. We list some of them with their *worst case running times* for **SEARCH, INSERT, DELETE**.

<table>
<thead>
<tr>
<th>DATA STRUCTURE</th>
<th>SEARCH</th>
<th>INSERT</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted singly linked list</td>
<td>$n$</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>unsorted doubly linked list</td>
<td>$n$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sorted array</td>
<td>log $n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>hash table</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>binary search tree</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>balanced search tree</td>
<td>log $n$</td>
<td>log $n$</td>
<td>log $n$</td>
</tr>
</tbody>
</table>

5 Binary Search Trees

**Definition.** For a node $x$ in a tree, $\text{height}(x)$ is equal to the length of the longest path from $x$ to a leaf.

**Definition.** For a node $x$ in a tree, $\text{depth}(x)$ is equal to the length of the path from $x$ to the root.

A binary tree is a *binary search tree* (BST) if it satisfies the BST Property

**BST Property.** For every node $x$, if node $y$ is in the left subtree of $x$, then $key(x) \geq key(y)$. If node $y$ is in the right subtree of $x$, then $key(x) \leq key(y)$. 


We will see why this property is useful for searching for a particular key. However, we will need to ensure that INSERT and DELETE maintain the BST Property. We will now consider a binary search tree as a data structure for the DICTIONARY ADT. We will begin by implementing SEARCH as follows:

Search (BST root R, key k):
   if R = null then
      return null
   else if ( k = key(R) ) then
      return R
   else if ( k < key(R) ) then
      return Search ( leftChild(R), k )
   else if ( k > key(R) ) then
      return Search ( rightChild(R), k )

In the worst case, we’ll start at the root of the tree and follow the longest path in the tree and then find that there is no node with key k. Since the length of the longest path in the tree is the definition of the height of the tree, this takes time $\Theta(\text{height of tree})$. For a tree with $n$ nodes, the height can be $n$ (if there are no right children, for instance)! So the worst-case running time (that is, for the worst tree and the worst $k$) is $\Theta(n)$.

Our implementation of INSERT follows:

Insert ( BST root R, node x ):
   if R = null then
      R := x
   else if ( key(x) < key(R) ) then
      Insert ( leftChild(R), x )
   else if ( key(x) > key(R) ) then
      Insert ( rightChild(R), x )
   else if ( key(x) = key(R) ) then
      /* depends on application */

   x will always be added as a leaf. Again we might have to follow the longest path from the root to a leaf and then insert $x$, so in the worst case, Insert takes time $\Theta(n)$.

The Delete operation is more complicated, so we describe it at a higher level.

Definition. succ($x$) is the node $y$ such that key($y$) is the lowest key that is higher than key($x$)

This definition of succ($x$) captures the intuitive notion of the successor of $x$.

Notice that if $x$ has a right child, then succ($x$) is the left-most node in the right subtree of $x$. In other words, starting from $x$’s right child, go left until there are no left children to follow. In this section, we will only call succ($x$) when $x$ has a right child.
Now, \texttt{Delete ( BST root R, node x )} has three cases:

1. If \( x \) has no children, simply remove it by setting \( x \) to \texttt{null}.

2. If \( x \) has one child \( y \) and \( z \) is the parent of \( x \), then we remove \( x \) and make \( y \) the appropriate child of \( z \) (i.e. the left child if \( x \) was the left child of \( z \) and the right child if \( x \) was the right child of \( z \)).

3. If \( x \) has two children, then let \( A \) and \( B \) be the left and right subtrees of \( x \), respectively. First we find \( \text{succ}(x) \). Then we set \( x \) to be \( \text{succ}(x) \) and \texttt{Delete \ succ(x)} (using either case 1 or case 2). By the definition of \( \text{succ}(x) \), we know that everything in \( A \) has key less than or equal to \( \text{key}(\text{succ}(x)) \). Let \( B' \) be \( B \) with \( \text{succ}(x) \) removed. Everything in \( B' \) must have key greater than or equal to \( \text{key}(\text{succ}(x)) \). Therefore, the \textbf{BST Property} is still maintained.

\textbf{Exercise.} Why is it guaranteed that deleting \( \text{succ}(x) \) always falls into case 1 or case 2 (i.e. case 3 never occurs when deleting \( \text{succ}(X) \))? 

Again, if \( x \) is the root and \( \text{succ}(x) \) is the leaf at the end of the longest path in the tree, then searching for \( \text{succ}(x) \) will take \( \Theta(\text{height of tree}) \) in the worst case. Since everything else we do takes constant time, the worst-case running time is \( \Theta(n) \) (Since in the worst case, the height of the tree is \( \Theta(n) \)).

Notice that the running times for these operations all depend on the height of the tree. If we had some guarantee that the tree’s height was smaller (in terms of the number of nodes it contains), then we would be able to support faster operations.

\section{6 Red-Black Trees}

A \textit{red-black tree} is a BST that also satisfies the following three properties:

1. Every node \( x \) is either red or black (\( \text{color}(x) = \text{red} \) or \( \text{color}(x) = \text{black} \)).

2. Both children of a red node are black.
3. For every node $x$, any path from $x$ to a descendant leaf contains the same number of black nodes.

**Definition.** For a node $x$ in a Red-Black tree, the black height or $BH(x)$ is the number of black nodes on a path between $x$ and a descendant leaf (not including $x$).

**Definition.** The black height of a tree $T$ ($BH(T)$) is the black height of its root.

Notice that this definition is well defined since the number of black nodes between $x$ and a leaf is always the same because of Property 3.

To make things work out easier, we’ll consider every node with a key value to be an internal node and the null values at the bottom of the tree will be the leaves and will be colored black.

**Example:**

```
          4
         /   \
        /     \
       2       5
      /   \   /   \   \
     1     3   5
   /   \   /   \   /   \   \
  h=0 h=1 h=2 h=0

BH = 0   BH = 0   BH = 0   BH = 0   BH = 0
```

6.1 Red-Black Trees Are Short

These three extra properties guarantee that the tree is approximately balanced and therefore, the height is bounded. More precisely:

**Theorem.** Any red-black tree with $n$ internal nodes has height at most $2 + 2 \log(n + 1)$.

To prove this theorem, we first prove the following lemma:

**Lemma.** For any node $x$ in a red-black tree, the number of nodes in the subtree rooted at $x$ is at least $2^{BH(x)} - 1$.

**Proof.** By induction on the height of $x$ (i.e. the length of the longest path from $x$ to a descendant leaf).
Base Case: The height of $x$ is 0.

Since the height of $x$ is 0, $x$ has no children. Certainly $BH(x)$ is 0 if its regular height is 0, so $x$ must have at least $2^0 - 1 = 0$ children. This is trivially true.

Inductive Step: Assume that the lemma is true for height of $x$ less than $n$. Prove that the lemma is true for height of $x$ equal to $n$.

We know that for any $x$ of height $n$, its children’s heights are less than $n$ and therefore the lemma is true for $x$’s children. Note that we are considering null to be a valid child. Each child $y$ must have black height at least $BH(x) - 1$ because if $y$ is black, it will have black height $BH(x) - 1$ and if $y$ is red, it will have black height $BH(x)$. Therefore, there must be at least $2^{BH(x) - 1} - 1$ internal nodes in the subtrees rooted at each child. Then, including $x$, we have at least $2(2^{BH(x) - 1} - 1) + 1 = 2^{BH(x)} - 1$ nodes in the subtree rooted at $x$.

Now we can easily prove the theorem:

Proof of Theorem. Let $h$ be the height of the tree. Property 2 says that on any path from the root to leaf, at least half of the nodes are black. So the black height of the root must be at least $\lceil h/2 \rceil - 1$ (since the root could be red). If $n$ is the number of internal nodes, then we know from the lemma that

\[
\begin{align*}
n &\geq 2^{\lceil h/2 \rceil - 1} - 1 \\
n + 1 &\geq 2^{\lceil h/2 \rceil - 1} \\
\log(n + 1) &\geq (\lceil h/2 \rceil - 1) \log 2 \\
1 + \log(n + 1) &\geq \lceil h/2 \rceil \\
1 + \log(n + 1) &\geq h/2 \\
2 + 2\log(n + 1) &\geq h
\end{align*}
\]
6.2 Search and Rotation on Red-Black Trees

We will now implement the three routines Search, Insert and Delete from the DICTIONARY ADT using the Red-Black Tree data structure.

Since a Red-Black Tree is a BST, we can use the same Search routine as before to search the tree in worst case time $\Theta(\log n)$ (since now the height of the tree is $\Theta(\log n)$ in the worst case). Insert and Delete will also take time $\Theta(\log n)$ but if we use the same routine as before, they will cause violations of one of the three Red-Black properties.

For instance, if we use the regular BST Insert, then we’ll add the new node at the bottom of the tree (so both its children are null). Then we have to decide whether to make it red or black. If we make it black, we’ll certainly violate property 3 of Red-Black trees. If we make it red, we don’t have to worry about property 3, but we might violate property 2 (if its parent is red).

The following two procedures will be useful in building our Insert and Delete methods:

\begin{verbatim}
RotateLeft(Tree T, Node x)
    Node y = rightChild(x);
    rightChild(x) = leftChild(y);
    leftChild(y) = x;

RotateRight(Tree T, Node y)
    Node x = leftChild(y);
    leftChild(y) = rightChild(x);
    rightChild(x) = y;
\end{verbatim}

These two methods perform what is referred to as a rotation on the tree $T$. The following is a graphical representation of these two methods. $x$ and $y$ are nodes and $A$, $B$, and $C$ are subtrees.

6.3 Insertion

We’ll use the following procedure to insert a node $x$ into a Red-Black tree:

\begin{verbatim}
RedBlackInsert(Tree Root R, Node x)
    Insert(R, x);
    color(x) = red;
\end{verbatim}
If property 2 is violated then
Fix the tree

Property 2 can be violated only if \( \text{parent}(x) \) is red. If \( \text{parent}(x) = R \), the root of the tree, then we can just recolor \( R \) to be black. This won’t violate property 3 for any node since there is nothing above \( R \). If \( \text{parent}(x) \neq R \), then we have three cases. We can assume \( \text{parent}(	ext{parent}(x)) \) is colored black since otherwise we would be violating property 2 before we even inserted \( x \).

We may need to apply the fixing operations multiple times before the tree is a proper Red-Black tree. Hence, even though \( x \) starts off with both children null, we might have moved it upwards in the tree using previous fixing operations, so \( x \) might in general have non-null children.

We will now consider the three cases. In each diagram, the objects shown are subtrees of the entire Red-Black tree. There may be nodes above it or nodes below it, except where otherwise specified. \( A \) and \( B \) are subtrees and \( w, x, y \) and \( z \) are single nodes. Squares represent black nodes and circles represent red nodes. In every case, we assume that the tree on the left does not violate property 3. Based on this assumption, you should check that the final subtree on the right also does not violate property 3.

Case 1 is the only case which leaves the tree on the right with a violation of Property 3. Cases 2 and 3 produce a proper Red-Black tree on the right without further iterations.

- **Case 1:** \( x \)’s “uncle” is red. (i.e. node \( w \) is red)

The problem here is that \( z \)’s parent might be red, so we still have a violation of property 2. But notice that we have moved the conflict upwards in the tree. Either we can keep applying case 1 until we reach the root, or we can apply case 2 or case 3 and end the fix up process. If we reach the root by applying case 1 (in other words, \( \text{parent}(z) \) is the root and \( \text{parent}(z) \) is red, then we just change \( \text{parent}(z) \) to black.

- **Case 2:** \( x \)’s uncle is not red (it’s black or does not exist) and \( \text{key}(x) \leq \text{key}(y) \leq \text{key}(z) \) (or \( \text{key}(x) \geq \text{key}(y) \geq \text{key}(z) \)).
Now there are no violations of either property 2 or property 3, so we are finished.

- **Case 3:** x’s uncle is not red and \(key(y) \leq key(x) \leq key(z)\) (or \(key(y) \geq key(x) \geq key(z)\)).

Now we can apply case 2 using \(y\) as \(x\) and we’re done.

6.3.1 Analysis

We know from our previous analysis that \textbf{Insert} takes worst case time \(\Theta(\log n)\). Now consider the running time of fixing the tree. In the worst case, we might have to apply case 1 until we move the red-red conflict all the way up to the root starting from the bottom of the tree. This takes time \(\Theta(\log n)\) since the height of the tree is \(\Theta(\log n)\). Combined with the \(\Theta(\log n)\) time to do the \textbf{Insert}, we find that \textbf{RedBlackInsert} takes time \(\Theta(\log n)\) in the worst case.

6.4 Deletion

It remains to be seen how Red-Black trees can support deletion in time \(\Theta(\log n)\), where \(n\) is the number of nodes in the tree. Recall that \textbf{RedBlackInsert} was essentially the same as the \textbf{Insert} operation for binary search trees except that it was followed by a “fix-up” process whenever the new node (which we colored red) had a red parent.

Recall that \textbf{Delete}(R, x) for BSTs deletes either node \(x\) if \(x\) has 0 or 1 children and \(\text{succ}(x)\) if \(x\) has 2 children. \textbf{RedBlackDelete} will be as follows:

\[
\text{RedBlackDelete(Tree Root R, Node x)}
\]

   \[
   \begin{align*}
   & \text{Delete}(R, x) \text{ but don’t actually delete } x, \text{ instead let } y \text{ be the } \\
   & \text{node that would have been deleted} \\
   & \text{If } y \text{ is red then} \\
   & \quad \text{Delete } y \\
   & \text{If } y \text{ is black then} \\
   & \quad \text{Fix the tree}
   \end{align*}
\]

If we perform \textbf{Delete}(R,x) on a Red-Black tree \(R\), then we remove a node \(y\) (which is either \(x\) or \(\text{succ}(x)\)). If \(y\) is red then we could not have possibly introduced any violations.

**Exercise.** Why could we have not introduced any violations when deleting a red node?
If \( y \) happens to be colored black, the black-height balance of the tree will almost certainly be upset and property 3 of Red-Black trees will be violated. So again, we will need a “fix-up” process. Recall also that \texttt{Delete} always removes a node that has at most one child (if \( x \) has two children, then we remove \( \text{succ}(x) \), which never has two children). Therefore, we have to worry about only those cases where \( y \) is black and \( y \) has at most one child.

- **Case A:** \( y \) has one child: \( y \)'s child, call it \( w \), must be red since otherwise property 3 would be violated in the subtree rooted at \( y \). So we can just remove \( y \) and make \( w \) black to preserve the black-height balance for any node above \( y \).

- **Case B:** \( y \) has no children: We can’t apply the above trick if \( y \) has no children. Recall that the \textit{null} values at the bottom of the tree are considered black leaves. To preserve the black-height for \( y \)'s ancestors, we’ll remove \( y \) and replace it with a \textit{null} node that is not just black, but “double-black” (denoted by a double circle). But while this upholds property 3 of Red-Black trees, it violates property 1 (that every node must be either red or black).

Now, we consider the problem of removing a double-black node from an arbitrary position in the tree. There are five cases for this; in all, \( r \) (which might be \textit{null}) will be the double-black node that we want to remove. You should check that the transformations preserve property 3 and do not introduce any property 2 violations.

- **Case 1:** \( r \)’s sibling is red: In this case, we modify the tree so that \( r \)'s neighbor is black and then apply one of the other cases. This is so that in general we can rely on the fact that \( r \)'s neighbor will be black:
Notice that this transformation has moved the double-black node downwards in the tree.

- **Case 2:** r’s parent, sibling and nephews are all black:

Notice that this transformation has moved the double-black node upwards in the tree. If this keeps happening, then eventually the root will become the double-black node and we can just change it to black without violating any properties. Otherwise we will be able to apply one of the other cases.

**Exercise.** Is it possible that Case 1 and Case 2 can conflict with each other by moving the double-black node downwards and then upwards in an infinite loop?

- **Case 3:** r’s sibling and nephews are black, r’s parent is red:

We can stop here because we have eliminated the double-black.

- **Case 4:** r’s far nephew is red: r’s parent s can start off as either color here (we’ll denote this by a rectangular node). After the transformation, t takes whatever color s had before.
Again, since there are no double-black nodes on the right, we can stop.

- **Case 5: r’s far nephew is black, r’s near nephew is red:** Here we’re going to perform a transformation so that r’s far nephew becomes red. Then we can apply Case 4.

### 6.4.1 Analysis

Again, we know that \texttt{Delete} takes $\Theta(\log n)$ time in the worst case. Now, consider the “fix-up” process.

Case A can be performed in constant time. Case B further breaks down into five cases. Case 1 moves the double black node down one position in the tree. Case 2 moves the double black node up one position in the tree. The other three positions eliminate the double black node in constant time.

If Case 1 is required, then the parent of the double black square is red. Therefore, one of Cases 3-5 are applied and the entire operation is performed in constant time.

Finally, Case 2 moves the double black node up one position in the tree and this will need to be performed $\Theta(\log n)$ times in the worst case. Therefore, the worst case running time of the “fix-up” operation is $\Theta(\log n)$.

Therefore, \texttt{RedBlackDelete} has a worst case running time of $\Theta(\log n)$. 


7 Augmenting Red-Black Trees

7.1 Introduction

Suppose that you are asked to implement an ADT that is the same as a dictionary but has one additional operation:

- operation: \( \text{SIZE(Set } S) \): Returns the current size of the set

If we try to implement this procedure without additional data in the data structure, the worst case running time would be \( \Theta(n) \) or worse.

But, if we add a size variable and increment it when we insert and decrement it when we delete, then the running time would be \( \Theta(1) \).

This is an example of augmenting a data structure.

7.2 Method

In this section we will look at three examples of augmenting red-black trees to support new queries. Any data structure can be augmented to provide additional functionality beyond the original ADT.

A red-black tree by itself is not very useful. All you can do is search the tree for a node with a certain key value. To support more useful queries we need to have more structure. When augmenting data structures, the following four steps are useful:

1. Pick a data structure to start with.
2. Determine additional information that needs to be maintained.
3. Check that the additional information can be maintained during each of the original operations (and at what additional cost, if any).
4. Implement the new operations.

7.3 Example 1

Let’s say we want to support the query \( \text{MIN}(R) \), which returns the node with minimum key-value in red-black tree \( R \).

One solution is to traverse the tree starting at the root and going left until there is no left-child. This node must have the minimum key-value. Since we might be traversing the height of the tree, this operation takes \( O(\log n) \) time.

Alternatively, we can store a pointer to the minimum node as part of the data structure (at the root, for instance). Then, to do a query \( \text{MIN}(R) \), all we have to do is return the pointer \( (R.min) \), which takes time \( O(1) \). The problem is that we might have to update the pointer whenever we perform \text{INSERT} or \text{DELETE}.

- \( \text{INSERT}(R, x) \): Insert \( x \) as before, but if \( \text{key}(x) < \text{key}(R.min) \), then update \( R.min \) to point at \( x \). This adds \( O(1) \) to the running time, so its complexity remains \( O(\log n) \).
DELETE(R, x): Delete x as before, but if \( x = R.min \), then update \( R.min \): if \( x \) was the minimum node, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is red (otherwise property 3 would be violated). This right-child is \( \text{succ}(x) \) and becomes the new minimum if it exists. If \( x \) had no children, then the new minimum is the parent of \( x \). Again we add \( O(1) \) to the running time of DELETE so it still takes \( O(\log n) \) in total.

This is the best-case scenario. We support a new query in \( O(1) \)-time without sacrificing the running times of the other operations.

7.4 Example 2

Now we want to know not just the minimum node in the tree, but, for any \( x \) in the tree, the minimum node in the subtree rooted at \( x \). We’ll call this query \( \text{SUBMIN}(R, x) \).

To achieve this query in time \( O(1) \), we’ll store at each \( x \) a pointer \( x.min \), to the minimum node in its subtree. Again, we’ll have to modify INSERT and DELETE to maintain this information.

- \( \text{INSERT}(R, x) \): Insert \( x \) as before, but, for each \( y \) that is an ancestor of \( x \), if \( \text{key}(x) < \text{key}(y.min) \), then update \( y.min \) to point at \( x \). This adds \( O(\log n) \) to the running time, so its complexity remains \( O(\log n) \).

- \( \text{DELETE}(R, x) \): Delete \( x \) as before, but, for each \( y \) that is an ancestor of \( x \), if \( x = y.min \), then update \( y.min \): if \( x \) was the minimum node in a subtree, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is red (otherwise property 3 would be violated). This right-child is \( \text{succ}(x) \) and becomes the new minimum if it exists. If \( x \) had no children, then the new minimum is the parent of \( x \). Again we add \( O(\log n) \) to the running time of DELETE so it still takes \( O(\log n) \) in total.

- Fix-up: The rotations (but not the recolorings) in the fix-up processes for INSERT and DELETE might affect the submins of certain nodes. Consider \( \text{RotateRight}(T, y) \) where \( x \) is the left child of \( y \). The submin of \( x \) will be in the subtree \( A \) (or will be \( x \) itself if \( A \) is empty). This doesn’t change after the rotation. The submin of \( y \), however, which used to be the same as \( x \)'s submin, is now in \( B \) (or \( y \) itself if \( B \) is empty). So, we set \( \text{SUBMIN}(R, y) \) to \( y \) if \( B \) is empty, or to \( \text{SUBMIN}(R, z) \), where \( z \) is the root of \( B \). It takes just constant time to reset \( \text{SUBMIN}(R, y) \), so rotations still take \( O(1) \). The modification for a left rotation are symmetric.

7.5 Example 3

We want to support the following queries:
• **RANK(R, k):** Given a key k, what is its "rank", i.e., its position among the elements in the red-black tree?

• **SELECT(R, r):** Given a rank r, what is the key with that rank?

**Example:**
If R contains the key-values 3,15,27,30,56, then \( \text{RANK}(R, 15) = 2 \) and \( \text{SELECT}(R, 4) = 30 \).

Here are three possibilities for implementation:

1. Use red-black trees without modification:
   - Queries: Simply carry out an inorder traversal of the tree, keeping track of the number of nodes visited, until the desired rank or key is reached. This requires time \( \Theta(n) \) in the worst case.
   - Updates: No additional information needs to be maintained.
   - Problem: Query time is very long and this method does not take advantage of the structure of the Red-Black tree. We want to be able to carry out both types of queries in only \( \Theta(\log n) \) time.

2. Augment red-black trees so that each node \( x \) has an additional field \( \text{rank}(x) \) that stores its rank in the tree.
   - Queries: Similar to \text{SEARCH}, choosing path according to key or rank field (depending on the type of query). This requires time \( \Theta(\log n) \), just like \text{SEARCH}.
   - Updates: Carry out normal update procedure, then update the \( \text{rank} \) field of all affected nodes. This can take time \( \Theta(n) \) in the worst case, since any insertion or deletion affects the rank of every node with higher key-value.
   - Problem: We’ve achieved the \( \Theta(\log n) \) query time we wanted, but at the expense of the update time, which has gone from \( \Theta(\log n) \) to \( \Theta(n) \). We would like all operations to have time at worst \( \Theta(\log n) \).

3. Augment red-black trees so that each node has an additional field \( \text{size}(x) \) that stores the number of nodes in the subtree rooted at \( x \) (including \( x \) itself).
   - Queries: We know that
     \[
     \text{rank}(x) = 1 + \text{number of nodes that come before } x \text{ in the tree}.
     \]

\( \text{RANK}(R, k) \): Given key \( k \), perform \text{SEARCH} on \( k \) keeping track of "current rank" \( r \) (which starts out as 0): when going left, \( r \) remains unchanged; when going right let \( r := r + \text{size}(\text{left}(x)) + 1 \). When \( x \) found such that \( \text{key}(x) = k \), output \( r + \text{size}(\text{left}(x)) + 1 \). Note that we did not deal with degenerate cases (such as when \( k \) does not belong to the tree), but it is easy to modify the algorithm to treat those cases.
SELECT\((R,r)\): Given rank \(r\), start at \(x = R\) and work down, looking for a node \(x\) such that \(r = \text{size}(\text{left}(x)) + 1\) (return that node once it is found). If \(r < \text{size}(\text{left}(x)) + 1\), then we know the node we are looking for is in the left subtree, so we go left without changing \(r\). If \(r > \text{size}(\text{left}(x)) + 1\), then we know the node we are looking for is in the right subtree, and that its relative rank in that tree is equal to \(r - (\text{size}(\text{left}(x)) + 1)\), so we change \(r\) accordingly and go right. Once again, we did not deal with degenerate cases (such as when \(r\) is a rank that does not correspond to any node in the tree), but they are easily accomodated with small changes to the algorithm.

- Query time: \(\Theta(\log n)\), as desired, since both algorithms are essentially like SEARCH (tracing a single path down from the root).

- Updates: INSERT and DELETE operations consist of two phases for red-black trees: the operation itself, followed by the fix-up process. We look at the operation phase first, and deal with the fix-up process afterwards.

INSERT\((R,x)\): We can set \(\text{size}(x) := 1\), and simply increment the \(\text{size}\) field for every ancestor of \(x\).

DELETE\((R,x)\): Consider the node \(y\) that is actually removed by the operation (so \(y = x\) or \(y = \text{succ}(x)\)). We know the size of the subtree rooted at every node on the path from \(y\) to the root decreases by 1, so we simply traverse that path to decrement the size of each node.

We’ve shown how to modify the INSERT and DELETE operations themselves. If we show how to do rotations and keep the \(\text{size}\) fields correct, then we’ll know how to do the whole fix-up process, since each case just consists of a rotation and/or a recoloring (recoloring does not affect the \(\text{size}\) field of any node).

Rotations: Consider right rotations (left rotations are similar).

\[
\begin{align*}
\text{size}(y) &= \text{size}(A) + \text{size}(B) + \text{size}(C) + 2 \\
\text{size}(x) &= \text{size}(A) + \text{size}(B) + 1
\end{align*}
\]

The only \(\text{size}\) fields that change are those of nodes \(x\) and \(y\), and the change is easily computed from the information available. So each rotation can be performed while maintaining the size information with only a constant amount of extra work.

- Update time: We have only added a constant amount of extra work during the first phase of each operation, and during each rotation, so the total time is still \(\Theta(\log n)\).

Now, we have finally achieved what we wanted: each operation (old or new) takes time \(\Theta(\log n)\) in the worst-case.
8 Direct Addressing

Recall that a dictionary is an ADT that supports the following operations on a set of elements with well-ordered key-values: INSERT, DELETE, SEARCH. If we know the key-values are integers from 1 to \(K\), for instance, then there is a simple and fast way to represent a dictionary: just allocate an array of size \(K\) and store an element with key \(i\) in the \(i\)th cell of the array.

This data structure is called direct addressing and supports all three of the important operations in worst-case time \(\Theta(1)\). There is a major problem with direct addressing, though. If the key-values are not bounded by a reasonable number, the array will be huge! Remember that the amount of space that a program requires is another measure of its complexity. Space, like time, is often a limited resource in computing.

Example 1: A good application of direct addressing is the problem of reading a text file and keeping track of the frequencies of each letter (one might need to do this for a compression algorithm such as Huffman coding). There are only 256 ASCII characters, so we could use an array of 256 cells, where the \(i\)th cell will hold the count of the number of occurrences of the \(i\)th ASCII character in our text file.

Example 2: A bad application of direct addressing is the problem of reading a data file (essentially a list of 32-bit integers) and keeping track of the frequencies of each number. The array would have to be of size \(2^{32}\), which is pretty big!

9 Hashing

A good observation about example 2 or about any situation where the range of key-values is large, is that a lot of these might not occur very much, or maybe even not at all. If this is the case, then we are wasting space by allocating an array with a cell for every single key-value.

Instead, we can build a hash table: if the key-values of our elements come from a universe (or set) \(U\), we can allocate a table (or an array) of size \(m\) (where \(m < |U|\)), and use a function \(h : U \rightarrow \{0, \ldots, m-1\}\) to decide where to store a given element (that is, an element with key-value \(x\) gets stored in position \(h(x)\) of the hash table). The function \(h\) is called a hash function.

Exercise. If the set of all keys was the set of all possible integer values (from 0 to \(2^{32} - 1\)), give some possible hash functions if \(m = 1,024\) (i.e. \(m = 2^{10}\)).

Definition. When two keys \(x \neq y\) hash to the same location (i.e. \(h(x) = h(y)\)), we say that they are in collision or that a collision has occurred.

Exercise. Would it be possible to set a hash function so that you could be sure you would have no collisions? How or why not?

Exercise. Consider a hash table where each location could hold \(b\) keys. Suppose we had \(b\) items already in a location (bucket) and another item \((b+1)\) hashed to the same location. What choices do we have about how to store this last item? Hint: Think about what you do in your personal phone book when you have too many friends whose name begin with the same letter (say "W").
9.1 Closed Addressing

If \( m < |U| \), then there must be \( k_1, k_2 \in U \) such that \( k_1 \neq k_2 \) and yet \( h(k_1) = h(k_2) \). This is called a collision; there are several ways to resolve it. One is to store a linked list at each entry in the hash table, so that an element with key \( k_1 \) and an element with key \( k_2 \) can both be stored at position \( h(k_1) = h(k_2) \) (see figure). This is called chaining.

Assuming we can compute \( h \) in constant time, then the \texttt{INSERT} operation will take time \( \Theta(1) \), since, given an element \( a \), we just compute \( i = h(\text{key}(a)) \) and insert \( a \) at the head of the linked list in position \( i \) of the hash table. \texttt{DELETE} also takes \( \Theta(1) \) if the list is doubly-linked (given a pointer to the element that should be deleted).

The complexity of \( \text{SEARCH}(S,k) \) is a little more complicated. If \( |U| > m(n-1) \), then any given hash function will put at least \( n \) key-values in some entry of the hash table. So, the worst case is when every entry of the table has no elements except for one entry which has \( n \) elements and we have to search to the end of that list to find \( k \) (see figure). This takes time \( \Theta(n) \) (not so good).

For the average case, the sample space is \( U \) (more precisely, the set of elements that have key-values from \( U \)). Whatever the probability distribution on \( U \), we assume that our hash function \( h \) obeys a property called simple uniform hashing. This means that if \( A_i \) is the event (subset of \( U \)) \( \{k \in U \mid h(k) = i\} \), then

\[
\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = \frac{1}{m}.
\]
In other words, each entry in the hash table is used just as much as any other. So the expected number of elements in any entry is \( n/m \). We will call this the \textit{load factor}, denoted by \( a \). This assumption may or may not be accurate depending on \( U \), \( h \) and the probability distribution on \( U \).

To calculate the average-case running time, let \( T \) be a random variable which counts the number of elements checked when searching for key \( k \). Let \( L_i \) be the length of the list at entry \( i \) in the hash table. Then the average-case running time is:

\[
E(T) = \sum_{k \in U} \Pr(k)T(k)
\]

\[
= \sum_{i=0}^{m-1} \sum_{k \in A_i} \Pr(k)T(k)
\]

\[
\leq \sum_{i=0}^{m-1} \Pr(A_i)L_i
\]

\[
= \frac{1}{m} \sum_{i=0}^{m-1} L_i
\]

\[
= \frac{n}{m}
\]

\[
= a
\]

So the average-case running time of \textsc{search} under simple uniform hashing with chaining is \( O(a) \). Depending on the application, we can sometimes consider \( a \) to be constant since we can make \( m \) bigger when we know that \( n \) will be large. When this is the case, \textsc{search} takes time \( O(1) \) on average.

\textbf{9.2 Examples of Hash Functions}

Recall the definition of simple uniform hashing: if \( A_i \) is the event (subset of \( U \)) \( \{ k \in U \mid h(k) = i \} \), then

\[
\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = 1/m.
\]

Basically, the hash table gets evenly used for whatever distribution of keys we are dealing with. The problem is that we often don’t know the distribution of keys before we see them. So how can we choose a good hash function?

For uniformly distributed keys in the range 1 through \( K \) (for large \( K \)), the following methods come close to simple, uniform hashing:

\textbf{The division method:} First choose a natural number \( m \). Then, the hash function is just

\[
h(k) = k \mod m.
\]

One advantage here is that computing \( h(k) \) is very fast (just one division operation). But \( m \) has to be chosen with some care. If \( m = 2^p \), then \( h(k) \) is just the \( p \) lowest bits of \( k \) (see example 1). Instead, \( m \) is usually chosen to be a prime not close to any power of 2.

\textbullet \textbf{Example:} Most compilers or interpreters of computer programs construct a symbol table to keep track of the identifiers used in the input program. A hash table is a good data structure
for a symbol table: identifiers need to be inserted and searched for quickly. We would like to use the division method for hashing, but first we need to turn the identifiers (strings of text) into positive integers. We can do this by considering each string to be a number in base 128 (if there are 128 text characters). Each character $x$ can be represented by a number from 1 through 128 denoted $num(x)$. Then, a string of characters $x_n x_{n-1} \ldots x_1$ can be represented uniquely by the number $\sum_{i=1}^{n} num(x_i)(128)^{i-1}$. For our choice of $m$ here, we definitely want to avoid powers of 2, especially powers of 128. If $m$ is $128^3$, for instance, then any two identifiers that share the same last three letters will hash to the same entry in the table. If the program is computing a lot of maximum values, for instance, then many of the variable names may end in “max” and they will all collide in the hash table, causing longer search times if we use chaining.

The multiplication method: Another way to hash natural numbers is just to scale them to something between 0 and $m-1$. Here we choose $m$ (often a power of 2 in this case) and a real number $A$ (often the fractional part of a common irrational number, such as the golden ratio: $(\sqrt{5} - 1)/2$). We then compute

$$h(k) = \lfloor m \times \text{frac}(kA) \rfloor,$$

where $\text{frac}(x)$ is the fractional part of a real number $x$.

Example

$\text{frac}(\frac{3}{2})$ is $\frac{1}{2}$ and $\text{frac}(1.77777\ldots)$ is 0.77777\ldots.

Exercise. What is the problem if $A$ is “very” rational, like $\frac{1}{2}$?

9.3 Open Addressing

In closed addressing, we handled collisions by enlarging the storage capacity at the relevant entry in the hash table (in particular, we did this using a linked-list). In open addressing, each entry in the hash table stores only one element (so, in particular, we only use it when $n < m$). If we try to insert a new element and we get collision, then we have to look for a new location to store the new element. But we have to put it somewhere where we can find it if we’re searching for it. To insert it, we check a well-defined sequence of other locations in the hash table until we find one that’s not full. This sequence is called a probe sequence. We will consider three different types of probe sequences.

1. Linear Probing: The easiest open addressing strategy is linear-probing. For a hash table of size $m$, key $k$ and hash function $h(k)$, the probe sequence is calculated as:

$$s_i = (h(k) + i) \mod m \quad \text{for } i = 0, 1, 2, \ldots.$$ 

Note that $s_0$ (the home location for the item) is $h(k)$ since $h(k)$ should map to a value between 0 and $m-1$.

Exercise. Work though an example where the $h(k) = k \mod 11$, $m = 11$, each bucket holds only one key and we use linear probing when collisions occur. Insert the keys 26, 21, 5, 36, 13, 16, 15 in that order.
Exercise. What is the problem with linear probing?

Exercise. How could we change the probing so that two items that hash to different home buckets don’t end up with nearly identical probe sequences?

Clustering: As soon as we hash to something within a group of filled locations, we have to probe the whole group until we reach an empty slot and in doing so we increase the size of the cluster. Two keys that didn’t necessarily share the same ”home” location end up with almost identical probe sequences.

2. Non-Linear Probing: Non-linear probing includes schemes where the probe sequence does not involve steps of fixed size. Consider quadratic probing where the probe sequence is calculated as:

\[ s_i = (h(k) + i^2) \mod m \quad \text{for } i = 0, 1, 2, \ldots. \]

There is still a problem, though: probe sequences will still be identical for elements that hash to the same home location.

Exercise. Work through an example where the \( h(k) = k \mod 11 \), \( n = 11 \), each bucket holds only one key and quadratic probing is used to resolve collisions. Insert the keys 26,21,5,36,13,16,15 in that order.

3. Double Hashing: In double hashing we use a different hash function \( h_2(k) \) to calculate the step size. The probe sequence is:

\[ A_i = (h(k) + i \cdot h_2(k)) \mod m \quad \text{for } i = 0, 1, 2, \ldots. \]

Note that \( h_2(k) \) shouldn’t be 0 for any \( k \). Also, we want to choose \( h_2 \) so that, if \( h(k_1) = h(k_2) \) for two keys \( k_1, k_2 \), it won’t be the case that \( h_2(k_1) = h_2(k_2) \). That is, the two hash functions don’t cause collisions on the same pairs of keys.

Exercise. Why is it important that \( h_2(k) \neq 0 \) for any \( k \)? What other choices for \( h_2(k) \) would be poor?

Analysis of Open Addressing: We’ll look at the complexity of \textsc{Insert} since, in open addressing, searching for a key \( k \) that is in the table takes exactly as long as it took to insert \( k \) in the first place. The time to search for an element \( k \) that does not appear in the table is the time it would take to insert that element in the table. You should check why these two statements are true.

It’s not hard to come up with worst-case situations where the above types of open addressing require \( \Theta(n) \) time for \textsc{Insert}. On average, however, it can be very difficult to analyze a particular type of probing. Therefore, we will consider the following situation: there is a hash table with \( m \) locations that contains \( n \) elements and we want to insert a new key \( k \). We will consider a random probe sequence for \( k \)—that is, it’s probe sequence is equally likely to be any permutation of \( (0, 1, \ldots, m-1) \). This is a realistic situation since, ideally, each key’s probe sequence is as unrelated as possible to the probe sequence of any other key.

Let \( T \) denote the number of probes performed in the \textsc{Insert}. Let \( A_i \) denote the event that every location up until the \( i \)-th probe is occupied. Then, \( T \geq i \) if \( A_1, A_2, \ldots, A_{i-1} \) all occur, so

\[
\Pr(T \geq i) = \Pr(A_1 \cap A_2 \cap \cdots \cap A_{i-1}) = \Pr(A_1) \Pr(A_2|A_1) \Pr(A_3|A_1 \cap A_2) \cdots \Pr(A_{i-1}|A_1 \cap \cdots \cap A_{i-2})
\]
For \( j \geq 1 \),

\[
Pr(A_j | A_1 \cap \cdots \cap A_{j-1}) = (n - j + 1)/(m - j + 1),
\]

because there are \( n - j + 1 \) elements that we haven’t seen among the remaining \( m - j + 1 \) slots that we haven’t seen. Hence,

\[
Pr(T \geq i) = \frac{n}{m} \cdot \frac{(n - 1)}{(m - 1)} \cdots \frac{(n - i + 2)}{(m - i + 2)} \leq \left( \frac{n}{m} \right)^{i-1} = a^{i-1}.
\] (13)

Now we can calculate the expected value of \( T \), or the average-case complexity of insert:

\[
E(T) = \sum_{i=0}^{m-1} i \Pr(T = i)
\]

\[
\leq \sum_{i=1}^{\infty} i \Pr(T = i)
\]

\[
= \sum_{i=1}^{\infty} i (\Pr(T \geq i) - \Pr(T \geq i + 1))
\]

\[
= \sum_{i=1}^{\infty} \Pr(T \geq i)
\]

\[
\leq \sum_{i=1}^{\infty} a^{i-1}
\]

\[
= \sum_{i=0}^{\infty} a^{i}
\]

\[
= \frac{1}{1 - a}
\]

Remember that \( a < 1 \) since \( n < m \). The bigger the load factor, however, the longer it takes to insert something. This is what we expect, intuitively.
10 Amortized Analysis

Often, we want to analyze the complexity of performing a sequence of operations on a particular data structure. In some cases, knowing the complexity of each operation in the sequence is important, so we can simply analyze the worst-case complexity of each operation. In other cases, only the time complexity for processing the entire sequence is important.

Definition. The worst-case sequence complexity of a sequence of \( m \) operations is the maximum total time over all sequences of \( m \) operations.

Notice that this is similar to the way that worst-case running time is defined. From this definition, it is obvious that the worst-case sequence complexity is less than or equal to \( m \) times the worst-case time complexity of a single operation in any sequence of \( m \) operations.

For example, suppose that we want to maintain a linked list of elements under the operations \textsc{INSERT}, \textsc{DELETE}, \textsc{SEARCH}, starting from an initially empty list. If we perform a sequence of \( m \) operations, what is the worst-case total time for all the operations? We know that the worst-case time for a single operation is \( \Theta(n) \) if the linked list contains \( n \) elements (\textsc{INSERT} and \textsc{DELETE} take time \( \Theta(1) \). \textsc{SEARCH} takes time \( \Theta(n) \)). Also, the maximum size of the linked list after \( n \) operations have been performed is \( n \). Hence, the worst-case running time of operation number \( i \) is simply \( i - 1 \) (the length of the list before operation \( i \)), so the worst-case sequence complexity of the \( m \) operations is at most

\[
\sum_{i=0}^{m-1} i = m(m-1)/2.
\]

We could have been a lot more careful about analyzing the situation, since \textsc{INSERT} runs in time \( O(1) \) and the only way the list can grow is by inserting elements. Hence, there must either be a lot of constant-time operations or we must have a pretty short list. This kind of insight, however, would complicate the analysis and would not lead to a better asymptotic value for the worst-case sequence complexity.

Definition. The amortized sequence complexity of a sequence of \( m \) operations is defined as follows:

\[
\text{amortized sequence complexity} = 1/m \times \text{the worst-case sequence complexity of the sequence } m
\]

Therefore, the amortized complexity represents the average worst-case complexity of each operation. But be careful: contrary to the average-case time complexity of one operation, the amortized complexity involves no probability. The average is simply taken over the number of operations performed.

Example
In our example above, the amortized sequence complexity is at most \( m(m-1)/2m = (m-1)/2 \).

Amortized analyses make more sense than a plain worst-case time analysis in many situations.
• A mail-order company employs a person to read customer’s letters and process each order: we care about the time taken to process a day’s worth of orders, for example, and not the time for each individual order.

• A symbol table in a compiler is used to keep track of information about variables in the program being compiled: we care about the time taken to process the entire program, i.e., the entire sequence of variables, and not about the time taken for each individual variable.

We will cover two basic methods for doing amortized analyses: the aggregate method and the accounting method. We’ve already seen an example of the aggregate method: simply compute the worst-case sequence complexity of the operations and divide by the number of operations in the sequence. We’re going to look at another example to illustrate both methods.

10.1 MULTIPOP

Suppose we want to extend the standard Stack ADT (that has operations \texttt{PUSH(S,x)} and \texttt{POP(S)} with a new operation \texttt{MULTIPOP(S,k)} that removes the top \(k\) elements from the stack. The time complexity of each \texttt{PUSH} and \texttt{POP} operation is \(\Theta(1)\), and the time complexity of \texttt{MULTIPOP(S,k)} is simply proportional to \(k\), the number of elements removed (actually, it’s proportional to \(\min(k, |S|)\), where \(|S|\) is the number of elements in stack \(S\)).

**The Aggregate Method:** In the aggregate method, we simply compute the worst-case sequence complexity of a sequence of operations and divide by the number of operations in the sequence.

For our \texttt{MULTIPOP} example, consider performing a total of \(n\) operations from among \texttt{PUSH}, \texttt{POP}, and \texttt{MULTIPOP}, on a stack that is initially empty. In this case, we could at first try to say that since the stack will never contain more than \(n\) elements, the cost of each operation is \(O(n)\), for a total of \(O(n^2)\). This gives us an average of \(O(n)\). In fact, we can do better if we realize that each object can be popped at most once for each time that it is pushed (including being popped by \texttt{MULTIPOP} operations). Since there can be at most \(n\) \texttt{PUSH} operations, there can be at most \(n\) \texttt{POP} operations (including counting the appropriate number \texttt{POP} operations for each \texttt{MULTIPOP}), which means that the total time taken for the entire sequence is at most \(O(n)\). This gives us that each operation takes on average \(O(1)\) time.

**The Accounting Method:** In the accounting method, we do the analysis as if we were an intermediate service providing access to the data structure. The \textit{cost} to us for each operation is the operation’s actual running time. We get to decide what we \textit{charge} the customer for each operation. Obviously, we want to cover our costs with what we earn in charges. Unlike a store, however, we want the total charge to be as close as possible to the total cost—this will give us the best estimate of the true complexity.

Typically we will charge more than the cost for some types of operations and charge nothing for other types. When we charge more than the cost, the leftover amount can be stored with the elements in the data structure as \textit{credit}. When we perform a “free” operation (i.e. no charge) on an element, we can use the credit stored with that element to pay for the cost of the operation.

If we assign charges and distribute credits carefully, we can ensure that each operation’s cost will be payed and that the total credit stored in the data structure is never negative. This indicates that the total amount charged for a sequence of operations is an upper bound on the total cost.
of the sequence, so we can use the total charge to compute an upper bound on the amortized complexity of the sequence.

For our MULTIPOP example, the cost of each operation (representing the time complexity of each operation) is as follows:

- \( \text{cost}(\text{PUSH}(S, x)) = 1 \)
- \( \text{cost}(\text{POP}(S)) = 1 \)
- \( \text{cost}(\text{MULTIPOP}(S, k)) = \min(k, |S|) \)

Since we know that each element can take part in at most two operations (one PUSH and one POP or MULTIPOP), the total "cost" for one element is 2, so we will assign charges as follows:

- \( \text{charge}(\text{PUSH}) = 2 \)
- \( \text{charge}(\text{POP}) = 0 \)
- \( \text{charge}(\text{MULTIPOP}) = 0 \)

This might seem strange at first, since we are charging nothing for POP or MULTIPOP, but it works out if we distribute credits appropriately. When an element is pushed onto the stack, we charge 2: 1 is used to pay for the cost of the PUSH, and 1 is assigned to the element as credit. When we POP an element from the stack, we charge nothing: the cost of the POP is payed for by using the credit of 1 that was stored with the element. Similarly, for MULTIPOP, the cost of removing each element can be payed for by using the credit stored with each element.

Since we’ve shown that each operation can be payed for, and since the total credit stored in the stack is never negative (each element has a credit of 1 while it is in the stack, and there can never be a negative number of elements in the stack), we have shown that the total charge for a sequence of \( m \) operations is an upper bound on the total cost for that sequence. But the total charge for \( m \) operations is at most \( 2m \), so the total cost is \( O(m) \). Dividing by the number of operations gives us an amortized complexity of \( O(1) \) for each operation.

10.2 Aggregate Method

When using the aggregate method, you can follow these steps:

1. State your costs as accurately as possible
2. Calculate a bound \( f(m) \) on those costs
3. Divide the bound \( f(m) \) by \( m \) to get a bound on the amortized sequence complexity

10.3 Accounting Method

When using the accounting method, you can follow these steps:

1. State your costs as accurately as possible
2. State what you are going to charge for each operation
3. State how and where your credits are going to be stored
4. State your credit invariant
5. Prove that your credit invariant is valid initially and across all possible operations
6. Show that you can always pay for each operation, based on the credit invariant
7. Calculate the amortized sequence complexity

10.4 Binary Counter

A Binary Counter is a sequence of \( k \) bits (\( k \) is fixed) on which a single operation can be performed: INCREMENT, which adds 1 to the integer represented in binary by the counter. The cost of a single INCREMENT operation is simply equal to the number of bits that need to be changed by the INCREMENT. For example, if \( k = 5 \),

\[
\begin{array}{l|l|l}
\text{Initial counter:} & 00000 & (\text{value} = 0) \\
\text{after INCREMENT:} & 00001 & (\text{value} = 1) \quad \text{cost} = 1 \\
\text{after INCREMENT:} & 00010 & (\text{value} = 2) \quad \text{cost} = 2 \\
\text{after INCREMENT:} & 00011 & (\text{value} = 3) \quad \text{cost} = 1 \\
\text{after INCREMENT:} & 00100 & (\text{value} = 4) \quad \text{cost} = 3 \\
\vdots & \\
\text{after INCREMENT:} & 11101 & (\text{value} = 29) \quad \text{cost} = 1 \\
\text{after INCREMENT:} & 11110 & (\text{value} = 30) \quad \text{cost} = 2 \\
\text{after INCREMENT:} & 11111 & (\text{value} = 31) \quad \text{cost} = 1 \\
\text{after INCREMENT:} & 00000 & (\text{value} = 0) \quad \text{cost} = 5 \\
\end{array}
\]

We can compute the amortized cost of a sequence of \( n \) INCREMENT operations, starting with value 0, as follows: Note that during the sequence of INCREMENT operations, we have the following situation (where we use the convention that bits of the counter are numbered from 0 (least significant bit) to \( k - 1 \) (most significant bit):

\[
\begin{array}{c|c|c}
\text{bit number} & \text{changes} & \text{total number of changes} \\
0 & \text{every operation} & n \\
1 & \text{every 2 operations} & \lfloor n/2 \rfloor \\
2 & \text{every 4 operations} & \lfloor n/4 \rfloor \\
\vdots & \\
i & \text{every } 2^i \text{ operations} & \lfloor n/2^i \rfloor \\
\end{array}
\]

Hence, the total number of bit-flips during the entire sequence is no more than the number of times bit \( i \) changes during the entire sequence, for bit numbers from 0 to \( \min\{k, \lfloor \log n \rfloor \} \) (the last bit that changes is bit number \( \lfloor \log n \rfloor \), except that if \( \log n > k \), there is no bit number \( \log n \)).
Hence, we get the following upper bound on the total number of bit-flips:

\[
\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\lfloor \log n \rfloor} \frac{n}{2^i} \leq n \sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n. \tag{14}
\]

This gives us an amortized cost of \(2n/n = 2\) for each operation in the sequence.

Let’s analyze the same problem using the accounting method instead of the aggregate method (which is what we did above, by finding the total cost directly). Consider what happens during one \texttt{INCREMENT} operation: a number of bits might be changed from 1 to 0 but exactly one bit will be changed from a 0 to a 1 (the rightmost bit with value 0).

For example, \texttt{INCREMENT(00111)} gives 01000, so three bits were changed from 1 to 0, but only one bit from 0 to 1. Hence, if we make sure that we have enough money stored in the counter to flip all the bits from 1 to 0, we can charge each operation only for the cost of flipping the 0 to a 1.

This is what we will do: even though the actual cost of an \texttt{INCREMENT} operation could be quite large, we charge each operation exactly 2: we use 1 to flip the 0 to a 1 and store the remaining 1 with the bit that was just changed to 1. Now, since we start the counter at 0, we can show the following credit invariant:

At any step during the sequence, each bit of the counter that is equal to 1 will have a credit of 1.

This can easily be proved by induction: initially, the counter is 0 and there is no credit, so the invariant is trivially true. Then, assuming that the invariant is true at a particular point, let’s perform one \texttt{INCREMENT} operation: the cost of flipping bits from 1 to 0 is payed for by the credit stored with each 1, the cost of flipping a single bit from 0 to 1 is payed for with 1 from the 2 charged to the operation, and we store the remaining 1 together with the bit that was just changed to 1. None of the other bits are changed. Hence, the credit invariant is still true (every bit equal to 1 has a 1 credit).

This shows that the total charge for the sequence of operations is an upper bound on the total cost of the sequence, and since in this case the total charge is \(2n\), we get that the amortized cost per operation is no more than \(2n/n = 2\) (same as before).
11 Dynamic Arrays

Consider the following data structure: we have an array of some fixed size, and two operations, APPEND (store an element in the first free position of the array) and DELETE (remove the element in the last occupied position of the array). This data structure is the standard way to implement stacks using an array.

It has one main advantage (accessing elements is very efficient), and one main disadvantage (the size of the structure is fixed). We can get around the disadvantage with the following idea: when trying to APPEND an element to an array that is full, first create a new array that is twice the size of the old one, copy all the elements from the old array into the new one, and then carry out the APPEND operation.

Let’s look at the cost of performing APPEND operations, starting from an array with size 0. We’ll only count the cost of assigning a value to an element of the array, disregarding the cost of allocating memory for each one (since most languages can usually allocate large chunks of memory efficiently, independently of the size of the memory required, and also because counting this cost would only add a constant factor more).

A: <Empty>

\[
\text{APPEND X1} \rightarrow \begin{array}{c}
X1 \\
\end{array} \quad \text{COST}=1
\]

\[
\text{APPEND X2} \rightarrow \begin{array}{c}
X1 \\
X2 \\
\end{array} \quad \text{COST}=2
\]

\[
\text{APPEND X3} \rightarrow \begin{array}{c}
X1 \\
X2 \\
X3 \\
\end{array} \quad \text{COST}=3
\]

\[
\text{APPEND X4} \rightarrow \begin{array}{c}
X1 \\
X2 \\
X3 \\
X4 \\
\end{array} \quad \text{COST}=1
\]

\[
\text{APPEND X5} \rightarrow \begin{array}{c}
X1 \\
X2 \\
X3 \\
X4 \\
X5 \\
\end{array} \quad \text{COST}=5
\]

\[
\text{APPEND X6} \rightarrow \begin{array}{c}
X1 \\
X2 \\
X3 \\
X4 \\
X5 \\
X6 \\
\end{array} \quad \text{COST}=1
\]

So generally, operation number \(i\) will cost 1, except if \(i = 2^k + 1\) for any natural number \(k\); then the cost will be \(i\).

We want to analyze the amortized complexity of a sequence of \(m\) APPEND operations, starting with an array of size 0. As in the binary counter example, let’s try charging 2 for each APPEND operation; then we should have enough to pay 1 for the cost of assigning the new element, and 1 to save with the element to pay for the cost of copying it later on.
A: <Empty>

\[
\begin{array}{c}
\xrightarrow{\text{APPEND X1}}& \begin{array}{c}
X1 \\
\end{array} & \text{COST=1 CHARGE=2 TOTAL CREDIT=1} \\
\xrightarrow{\text{APPEND X2}}& \begin{array}{c}
X1 X2 \\
\end{array} & \text{COST=2 CHARGE=2 TOTAL CREDIT=1} \\
\xrightarrow{\text{APPEND X3}}& \begin{array}{c}
X1 X2 X3 \\
\end{array} & \text{COST=3 CHARGE=2 TOTAL CREDIT=0} \\
\xrightarrow{\text{APPEND X4}}& \begin{array}{c}
X1 X2 X3 X4 \\
\end{array} & \text{COST=1 CHARGE=2 TOTAL CREDIT=1} \\
\xrightarrow{\text{APPEND X5}}& \begin{array}{c}
X1 X2 X3 X4 X5 \\
\end{array} & \text{COST=5 CHARGE=2 TOTAL CREDIT=-2}
\end{array}
\]

As you can see, we run into a problem: we don’t have enough credits to copy over all of the old elements! In fact, what ends up happening is that only the elements in the second half of the array (the ones added since the last size increase) have a credit on them. This suggests the following solution to our problem: make each element in the second half responsible for paying to copy both itself and one other element from the first half of the array.

So, if we charge 3 for each APPEND operation, we can prove the following credit invariant: Each element in the second half of the array has a credit of 2.

**Proof of credit invariant:** Initially, the array has size 0 and no elements, so the invariant is trivially true. Assume that the invariant is true after a certain number of APPEND operations have been performed, and consider the next APPEND operation:

- If the size of the array does not need to be changed, simply use 1 to pay for storing the new element, and keep 2 as credit with that new element. Since new elements are only added in the second half of the array, the credit invariant is maintained.

- If the size of the array needs to be changed, then this means that the array is full. Since the number of elements in the first half of the array is the same as the number of elements in the second half of the array, and since we have 2 credits on each element in the second half, we have exactly enough money to pay for the cost of copying all the elements into the new array of twice the size. Then, we use the 3 as before, to pay for storing the new element and keep 2 credits on that new element. As before the invariant is maintained.

Hence, the number of credits in the array never becomes negative, so the total charge for the sequence is an upper bound on the total cost of the sequence, i.e., the sequence complexity of \(m\) APPEND operations is at most \(3m\) and the amortized cost of a single operation in this sequence is \(3m/m = 3\).
If we look at what happens when we include DELETE operations, we can simply charge each DELETE operation 1 to pay for the cost of removing one element. Notice that this does not affect the credit invariant. Now, since we charge 3 for the most expensive operation (namely APPEND), the worst-case sequence complexity of \( m \) operations is \( 3m \) and the amortized sequence complexity is 3.

12 Dynamic Arrays with Reducing

If many DELETE operations are performed, the array could become very empty, which wastes memory space. We would like to contract the size of the array when it becomes "too empty", which involves creating a new array with a smaller size and copying every element over into the new array. Consider the following policy:

Suppose that we reduce the size of the array in half when a DELETE operation causes the array to become less than half full. Unfortunately, this leads to the following situation: consider a sequence of \( n = 2^k \) operations, where the first \( n/2 \) operations are APPEND, followed by the sequence APPEND, DELETE, DELETE, APPEND, APPEND, DELETE, DELETE, .... The first APPEND in the second half of the sequence of operations will cause the array to grow (so \( n/2 \) elements need to be copied over), while the two DELETE operations will cause the new array to become less than half full, so that it shrinks (copying \( n/2 - 1 \) elements), the next two APPEND operations cause it to grow again (copying \( n/2 \) elements), etc. Hence, the total cost for this sequence of \( n \) operations is \( \Omega(n^2) \), which gives an amortized cost of \( n \).

Intuitively, we need to perform more deletions before contracting the array size. Consider what happens if we wait until the array becomes less than 1/4 full before reducing its size by half.

\[
\begin{array}{cccc}
X1 & X2 & & \\
\end{array}
\]

DELETE X2

\[
\begin{array}{cc}
X1 & \\
\end{array}
\]

Then, no matter how many elements the array had to start with, we must delete at least 1/4 of the elements in the array before a contraction occurs, and once a contraction occurs, we must add at least as many elements as there are left before an expansion occurs. This gives us enough time to amass credit, and to maintain the following two-part credit invariant:

1. Every element in the second half of the array has credit 2 (this is the same as the non-reducing case).

2. If the array is less than half full, the amount of credit in the first quarter of the array is at least the number of elements by which the array is less than half full.
Basically, as we delete elements from the array and thereby get closer to reducing it, we want to build up credit in the first quarter of the array because these are the elements that will need to be copied during the reduction.

To achieve this credit invariant, we use the following charging scheme: we charge 3 for \texttt{APPEND} and 2 for \texttt{DELETE}.

\textbf{Proof of credit invariant:}

- **Base Case:** Initially, the array is empty and has size 0, so the credit invariant is vacuously true.

- **Inductive Step:** After a certain number of operations have been performed, assume that the credit invariant holds. Now consider the next operation:
  - **Case A:** If \texttt{APPEND} is performed, treat it just like before (if the array is full, there are enough credits to double the size and copy all the elements over, and of the charge of 3, 1 pays for the new element and 2 stays as credit with the new element).
  - **Case B:** If \texttt{DELETE} is performed, there are three cases to consider:
    * **Case B1:** If the element deleted is in the second half of the array, we pay for the deletion using 1 of the 2 charged and simply "throw away" the remaining 1 as well as the 2 credits that were stored with the element.
    * **Case B2:** If the element deleted is in the first half of the array but not in the first quarter, then we pay for the cost of the deletion using 1 and put the other 1 as credit on the first element in the array that has 0 credit (if there is no such element, we just "throw away" the extra 1).
    * **Case B3:** If the element deleted is in the first quarter, it must be the last element in the first quarter of the array, so by the credit invariant, every element in the first quarter has at least 1 credit: we use those credits to pay for the cost of copying each element into a new array of half the size. The 1 that was stored with the deleted element can be used to pay for the \texttt{DELETE}. That leaves 2 more from the charge of \texttt{DELETE}. We give one of these to the first element in the array if it has no credits. Otherwise we just throw them out. Note that the array is now one element short of half full. In accordance with the credit invariant, we have made sure that there is at least 1 credit in the first quarter.

In all cases, the credit invariant is maintained, so the total credit of the data structure is never negative, meaning that the total charge is an upper bound on the total cost of the sequence of
operations. Since the total charge for $m$ operations is $\leq 3m$, the amortized cost of each operation is $\leq 3m/m = 3$.

Notice that in this case, we really are overcharging for some of the operations (because we sometimes throw away credits that are not needed), but this is necessary if we want to ensure that we always have enough credits in all possible cases.
13 Graphs

A graph $G = (V, E)$ consists of a set of vertices (or nodes) $V$ and a set of edges $E$. In general, we let $n = |V|$, the number of nodes, and $m = |E|$, the number of edges. In a directed graph, each edge is an ordered pair of nodes $(u, v)$ (so $(u, v)$ is considered different from $(v, u)$); also, self-loops (edges of the form $(u, u)$) are allowed. In an undirected graph, each edge is a set of two vertices $\{u, v\}$ (so $\{u, v\}$ and $\{v, u\}$ are the same), and self-loops are disallowed. In a weighted graph each edge $e \in E$ is assigned a real number $w(e)$ called its weight.

An undirected graph is said to be **connected** if there is a path between every two vertices. A directed graph is said to be **strongly connected** if, for any two vertices $u, v$, there is a directed path from $u$ to $v$.

Some standard operations on graphs are:

- **Add a vertex; Remove a vertex; Add an edge; Remove an edge.**
- **Edge Query**: given two vertices $u, v$, find out if the edge $(u, v)$ (if the graph is directed) or the edge $\{u, v\}$ (if it is undirected) is in the graph.
- **Neighborhood**: given a vertex $u$ in an undirected graph, get the set of vertices $v$ such that $\{u, v\}$ is an edge.
- **In-neighborhood (out-neighborhood)**: given a vertex $u$ in a directed graph, get the set of vertices $v$ such that $(v, u)$ (or $(u, v)$, respectively) is an edge.
- **Degree, in-degree, out-degree**: compute the size of the neighborhood, in-neighborhood, or out-neighborhood, respectively.
- **Traversal**: visit each vertex of a graph to perform some task.
13.1 Data structures for graphs

There are two standard data structures used to store graphs: adjacency matrices, and adjacency lists.

- For an adjacency matrix, let $V = \{v_1, v_2, ..., v_n\}$. Then, we store information about the edges of the graph in an $n \times n$ array $A$ where

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \quad (18)$$

For undirected graphs, the matrix will be symmetric ($A[i, j]$ and $A[j, i]$ will always hold the same value). This requires space $\Theta(n^2)$ but edge queries are $\Theta(1)$. If the graph is weighted, we let $A[i, j]$ store the weight of the edge $(v_i, v_j)$ if that edge exists, and either 0 or $\infty$ if the edge doesn’t exist, depending on the application.

- For an adjacency list, we have a 1-dimensional array $A$ of size $n$. At entry $A[i]$, we store a linked-list of neighbors of $v_i$ (if the graph is directed, we store only the out-neighbors).

The amount of storage required is $\Theta(n+m)$ since each edge $(v_i, v_j)$ of the graph is represented by exactly one linked-list node in the directed case (namely, the node storing $v_j$ in the linked list at $A[i]$), and by exactly two linked-list nodes in the undirected case (node $v_j$ in the list at $A[i]$ and node $v_i$ in the list at $A[j]$). Edge queries can be made $\Theta(\log n)$ (actually, $\Theta(\log(\text{maximum degree}))$) if the lists are stored as balanced trees.

We now examine two ways to traverse a graph:

14 Breadth-First Search (BFS)

BFS takes a graph given as an adjacency list. Starting from a specified source vertex $s \in V$, BFS visits every vertex $v \in V$ that can be reached from $s$, and keeps track of the path from $s$ to $v$ with the smallest number of edges. BFS works on directed or undirected graphs: we describe it for directed graphs.

To keep track of progress, each vertex is given a color, which is initially white. The first time that a vertex is encountered, its color is changed to gray. When we finish with a vertex, its color is changed to black. At the same time, for each vertex $v$, we also keep track of the predecessor of $v$ in the BFS tree, $p[v]$, and we keep track of the number of edges from $s$ to $v$, $d[v]$.

In order to work in a “breadth-first” manner, BFS uses a first-in, first-out (FIFO) queue $Q$ to store the vertices. $Q$ has operations ENQUEUE($Q$, $v$), DEQUEUE($Q$) and ISEMPTY($Q$).

BFS($G=(V,E), s$)
  for all vertices $v$ in $V$
    color[$v$] := white
    $d[v] := \infty$
    $p[v] := \text{NIL}$
  end for
  initialize an empty queue $Q$;
  color[$s$] := gray;

39
d[s] := 0;
p[s] := NIL;
ENQUEUE(Q,s);
while not ISEMPTY(Q) do
    u := DEQUEUE(Q);
    for each edge (u,v) in E do
        if (color[v] == white) then
            color[v] := gray;
            d[v] := d[u] + 1;
            p[v] := u;
            ENQUEUE(Q,v);
        end if
    end for
    color[u] := black;
end while
END BFS

Each node is ENQUEUEed at most once, since a node is ENQUEUEed only when it is white, and its
color is changed the first time it is ENQUEUEed. In particular, this means that the adjacency list of
each node is examined at most once, so that the total running time of BFS is $O(n + m)$, linear in
the size of the adjacency list.

Notice that BFS will visit only those vertices that are reachable from $s$. If the graph is connected
(in the undirected case) or strongly-connected (in the directed case), then this will be all the vertices.
If not, then we may have to call BFS on more than one start vertex in order to see the whole graph.

For a proof that $d[v]$ really does represent the length of the shortest path (in terms of number
of edges) from $s$ to $v$, consult the text.

Below is a graph showing possible values for $d$ and $p$. Note that BFS might assign these values
slightly differently depending on the order in which the neighbors of each vertex are listed in the
adjacency list.

15 Depth-First Search

Just like for BFS, each vertex will be colored white (when it hasn’t been ”discovered” yet), gray
(when it’s been encountered but its adjacency list hasn’t been completely visited yet), or black
(when its adjacency list has been completely visited). The philosophy of DFS is ”go as far as
possible before backtracking”, so we will also keep track of two "timestamps" for each vertex: \(d[v]\) will indicate the discovery time (when the vertex was first encountered) and \(f[v]\) will indicate the finish time (when it’s been completely visited).

In order to implement the “depth-first” strategy, DFS uses a stack \(S\) to store edges. \(S\) will have operations \(\text{PUSH}(S,(u,v))\), \(\text{POP}(S)\), \(\text{ISEMPTY}(S)\). For this algorithm, we will need a function \(\text{timeStamp}()\) which returns 0 the first time that it is called and on each subsequent call, it returns the next largest integer.

\[
\text{DFS}(G=(V,E),s) \\
\text{for all vertices } v \text{ in } V \\
\quad \text{color}[v] := \text{white}; \\
\quad d[v] := \infty; \\
\quad f[v] := \infty; \\
\quad p[v] := \text{NIL}; \\
\text{end for} \\
\text{initialize an empty stack } S; \\
\text{color}[s] := \text{gray}; \\
\quad d[s] := \text{timeStamp}(); \\
\quad p[s] := \text{NIL}; \\
\quad \text{PUSH}(S,(s,\text{NIL})); \\
\text{for each edge } (s,v) \text{ in } E \text{ do} \\
\quad \text{PUSH}(S,(s,v)); \\
\text{end for} \\
\text{while not ISEMPTY}(S) \text{ do} \\
\quad (u,v) := \text{POP}(S); \\
\quad \text{if } (v == \text{NIL}) \text{ then} \quad \text{// Done with } u \\
\quad\quad \quad f[u] := \text{timeStamp}(); \\
\quad\quad \quad \text{color}[u] := \text{black}; \\
\quad\text{else if } (\text{color}[v] == \text{white}) \text{ then} \\
\quad\quad \quad \text{color}[v] := \text{gray}; \\
\quad\quad \quad d[v] := \text{timeStamp}(); \\
\quad\quad \quad p[v] := u; \\
\quad\quad \quad \text{PUSH}(S,(v,\text{NIL})); \quad \text{// Marks the end of } v \text{'s neighbors} \\
\quad\quad \text{for each edge } (v,w) \text{ in } E \text{ do} \\
\quad\quad\quad \text{PUSH}(S,(v,w)); \\
\quad\text{end for} \\
\text{(*) end if} \\
\text{end while} \\
\text{END DFS}
\]

Since DFS visits the neighbors of a node only when that node is white, vertices become gray the first time they are visited and for each vertex we visit its adjacency list at most once, the total running time is \(\Theta(n + m)\) (linear in the size of the adjacency list). As with BFS, DFS will visit only those vertices that are reachable from \(s\).

Below is a graph showing possible values for \(d, p\) and \(f\). Again, DFS might assign these values slightly differently depending on the order of the adjacency lists.
Note that DFS constructs a "DFS-tree" for the graph, by keeping track of a predecessor $p[v]$ for each node $v$. For certain applications, we need to distinguish between different types of edges in $E$:

- **Tree Edges** are the edges in the DFS tree.
- **Back Edges** are edges from a vertex $u$ to an ancestor of $u$ in the DFS tree.
- **Forward Edges** are edges from a vertex $u$ to a descendent of $u$ in the DFS tree.
- **Cross Edges** are all the other edges that are not part of the DFS tree (from a vertex $u$ to another vertex $v$ that is neither an ancestor nor a descendent of $u$ in the DFS tree).

The following diagram gives one possible output for DFS and labels the types of edges (Note that this graph is not the same underlying graph as the previous graph):

One application of DFS is determining whether a graph $G$, given as an adjacency matrix, has any cycles in it. A cycle in a graph is a path from a vertex $u$ to itself. It is not hard to see that there is a cycle in $G$ if and only if there are any back edges when DFS is run. To detect a back edge during the execution of DFS, we can add a test after the line marked by (*) in DFS. If the color of $v$ is gray instead of white, then we know that we have seen $v$ before on the current path from the source $s$. This means that the edge $(u, v)$ is a back edge and therefore forms a cycle.
16 Minimum Cost Spanning Trees (MCSTs)

Let $G = (V, E)$ be a connected, undirected graph with edge weights $w(e)$ for each edge $e \in E$. A tree is a subset of edges $A \subset E$ such that $A$ is connected and contains no cycles. The following diagram shows a graph with three different subsets $A$ (the thick edges are in $A$, the thin ones are not). One is a tree and the other two aren’t.

A spanning tree is a tree $A$ such that every vertex $v \in V$ is an endpoint of at least one edge in $A$. Notice that any spanning tree must contain $n - 1$ edges, where $|V| = n$ (proof by induction on $n$).

A minimum cost spanning tree is a spanning tree $A$ such that

$$w(A) = \sum_{e \in A} w(e)$$
is less than or equal to $w(B)$, for all other spanning trees $B$.

Minimum Cost Spanning Tree

Spanning tree, but not minimum cost

Two different MCSTs on the same graph

Finding MCSTs is important in practice: imagine you have a network of computers that are connected by various links. Some of these links are faster, or more reliable, than others. You might want to pick a minimal set of links that connects every computer (in other words, a spanning tree) such that these links are overall the best (they have minimum cost). Once you have found these links, you never have to use the remaining slower, or less reliable, links.

We will look at two algorithms for constructing MCSTs. The first is Prim’s Algorithm.

16.1 Prim’s Algorithm

Prim’s algorithm uses a Priority Queue ADT. This operates on a set $S$ where each element $x \in S$ has a priority $p(x)$ which comes from a well-ordered universe (usually the natural numbers). There are three operations on this set:

- $\text{INSERT}(S,x)$: insert an element $x$ in the set $S$.
- $\text{ISEMPTY}(S)$: return true if $S$ is empty.
- $\text{EXTRACT-MIN}(S)$: remove and return an element $x \in S$ with minimum priority.

In addition, we will need the operation $\text{DECREASE-_PRIORITY}(x, p)$ which sets the priority of $x$, which is in the queue, to be $p$ ($p$ is less than $x$’s current priority).

$$\text{PRIM-MST}(G=(V,E),w:E\rightarrow\mathbb{Z})$$
$$A := \{\};$$
$$\text{initialize a priority queue } Q;$$
$$\text{for all } v \text{ in } V \text{ do}$$
priority[v] := infinity;
p[v] := NIL;
INSERT(Q,v);
pick some arbitrary vertex s in V and let priority[s] := 0;
for each v in adjacency-list[s] do
  if v in Q and w(u,v) < priority[v] then
    DECREASE-PRIORITY(v, w(s, v))
p[v] := s;
while ( not ISEMPTY(Q) ) do
  u := EXTRACT-MIN(Q);
  A := A U {(p[u], u)};
  for each v in adjacency-list[u] do
    if v in Q and w(u,v) < priority[v] then
      DECREASE-PRIORITY(v, w(u,v));
p[v] := u;
  end if
end for
end while
END PRIM-MST

Prim’s algorithm grows an MCST A starting with an empty set. Even though A is technically a set of edges, we can consider the vertices in A as the set of vertices that are the endpoints of edges in A. When we start the algorithm, however, we consider the vertex s to be in A even though there are no edges in A. From now on, we just keep adding edges to A by finding the “lightest” edge that has one endpoint in A and the other endpoint outside of A.

16.2 Correctness

The correctness of Prim’s algorithm is given by the following theorem.

**Theorem:** If $G = (V, E)$ is a connected, undirected, weighted graph, A is a subset of some MCST of G, and $e$ is any edge of minimum weight with one endpoint in A and one endpoint outside of A, then $A \cup \{e\}$ is a subset of some MCST of $G$.

**Proof:** Let $T$ be a MCST of $G$ that contains $A$ as a subset. If $e$ is in $T$, then we are done. Otherwise, we construct a different MCST $T'$ that contains $e$. If we add edge $e$ to $T$, we create a cycle in the resulting graph ($T \cup \{e\}$ is not a tree anymore). This cycle must contain another edge $e'$ with one endpoint in $A$ and one endpoint outside of $A$ (otherwise, every endpoint of $A$ is already in $T$ which means that edge $e$ cannot exist). Since we picked $e$ to have minimum weight among such edges, it must be the case that $w(e) \leq w(e')$. Now, let $T' = T \cup \{e\} - \{e'\}$. $T'$ is connected, and it is acyclic (since we removed one edge from the only cycle in $T \cup \{e\}$), so it is a spanning tree of $G$ that contains $A$ as a subset. Moreover, $w(T') - w(T) = w(e) - w(e') \leq 0$ so $T'$ has total weight no greater than $T$, i.e., $T'$ is an MCST that contains $A \cup \{e\}$ as a subset.
17 Priority Queues

Priority queues are very useful. Some of their applications are:

- Job scheduling in operating systems
- Printer queues
- Event-driven simulation algorithms
- Greedy algorithms

There are several possible data structures for implementing priority queues:

- Unsorted list: takes time $\Theta(n)$ for $\text{EXTRACT-MIN}$ in the worst-case.
- Sorted list (by priorities): takes time $\Theta(n)$ for $\text{INSERT}$ in worst-case.
- Red-Black tree (key-values are priorities): $\text{INSERT}$ and $\text{EXTRACT-MIN}$ take time $\Theta(\log n)$.
- Direct addressing: if the universe $U$ of priorities is small and the priorities are all distinct, then we can store an element with priority $k$ in the $k$th cell of an array. $\text{INSERT}$ takes time $\Theta(1)$. $\text{EXTRACT-MIN}$ requires time $\Theta(|U|)$ in the worst-case (have to look at each location to find the first nonempty one).

18 Heaps

We will look at one particular data structure for priority queues in depth. They are called heaps and are defined as follows: a heap is a binary tree $T$ of elements with priorities such that

1. $T$ is complete: this means that every level of the tree is full except perhaps the bottom one, which fills up from left to right. For example:

![Complete, Not complete, Not complete trees]

2. For each node $x$ in $T$, if $x$ has a left-child, then $p(x) \leq p(\text{left}(x))$ and if $x$ has a right-child, then $p(x) \leq p(\text{right}(x))$.

We can conclude a few immediate facts about heaps from the definition. First of all, the root has minimum priority. Secondly, every subtree of a heap is also a heap (in particular, an empty tree is a heap). Finally, since heaps are complete, if a heap contains $n$ nodes, then its height $h$ is $\Theta(\log n)$. 
18.1 Storing heaps


$$A = [4,6,8,7,7,9,12,13,14]$$

If the size of the array is close to the number of elements in the heap, then this data structure is extremely space-efficient because we don’t have to store any pointers. We can use a dynamic array to ensure that this is true (recall that the amortized cost of managing a dynamic array is small).

18.2 Implementing priority queues

We can perform the priority queue operations on a heap as follows:

- **INSERT**: Increment $heapsize$ and add the new element at the end of the array. The result might violate the heap property, so ”percolate” the element up (exchanging it with its parent) until its priority is no smaller than the priority of its parent.

For example, if we perform INSERT(5) on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):
In the worst-case, we will have to move the new element all the way to the root, which takes time $\Theta(\text{height of heap}) = \Theta(\log n)$.

- **EXTRACT-MIN**: Decrement $\text{heapsize}$ and remove the first element of the array. In order to be left with a valid heap, move the last element in the array to the first position (so the heap now has the right "shape"), and percolate this element down until its priority is no greater than the priorities of both its children. Do this by by exchanging the element with its child of lowest priority at every step.

For example, if we perform EXTRACT-MIN on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):
As with INSERT, we may wind up moving the last element from the root all the way down to a leaf, which takes $\Theta(\text{height of heap}) = \Theta(\log n)$ in the worst-case.

The "percolating down" of an element that we just described for EXTRACT-MIN is a very useful operation for heaps. In fact, it’s so useful that it already has a name: If $x$ is the element initially stored at $A[i]$, and assuming that the left and right subtrees of $x$ are heaps. Then HEAPIFY($A, i$) percolates $x$ downwards until the subtree of the element now stored at $A[i]$ is a heap.

HEAPIFY($A, i$)
smallest := i;
if ( 2i <= heapsize and A[2i] < A[i] ) then
    smallest := 2i;
endif
if ( 2i+1 <= heapsize and A[2i+1] < A[smallest] ) then
    smallest := 2i+1;
endif
if( smallest != i ) then
    swap A[i] and A[smallest];
    HEAPIFY(A,smallest);
endif
END

The running time, as with EXTRACT-MIN, is $\Theta(\log n)$.

### 18.3 Building heaps

If we start with an array $A$ of elements with priorities, whose only empty slots are at the far right, then we can immediately view $A$ as a complete binary tree. $A$, however, is not necessarily a heap unless the elements are ordered in a certain way. There are several options for making $A$ into a heap:

1. Sort $A$ from lowest priority element to highest. Clearly $A$ will now obey part 2 of the heap definition (actually, every sorted array is a heap, but every heap is not necessarily a sorted array). This takes time $\Theta(n \log n)$ if we use, say, the guaranteed fast version of quicksort.

2. We can simply make a new array $B$ and go through every element of $A$ and INSERT it into $B$. Since INSERT takes time $\Theta(\log n)$ and we do it for each of the $n$ elements of $A$, the whole thing takes time $\Theta(n \log n)$.

3. The most efficient way is to use HEAPIFY: notice that every item in the second half of $A$ corresponds to a leaf in the tree represented by $A$, so starting at the "middle" element (i.e., the first nonleaf node in the tree represented by $A$), we simply call HEAPIFY on each position of the array, working back towards position 1.

BUILD-HEAP($A$)
    heapsize := size($A$);
    for i := floor(heapsize/2) downto 1 do
        HEAPIFY($A$,i);
    end for
END

Because each item in the second half of the array is already a heap (it’s a leaf), the preconditions for HEAPIFY are always satisfied before each call. For example, if $A = [1,5,7,6,2,9,4,8]$, then BUILD-HEAP($A$) makes the following sequence of calls to HEAPIFY (you can check the result of each one by tracing it):
\[
\text{HEAPIFY( } [1,5,7,6,2,9,4,8], 4 \text{ ) } = [1,5,7,6,2,9,4,8]
\]
\[
\text{HEAPIFY( } [1,5,7,6,2,9,4,8], 3 \text{ ) } = [1,5,4,6,2,9,7,8]
\]
\[
\text{HEAPIFY( } [1,5,4,6,2,9,7,8], 2 \text{ ) } = [1,2,4,6,5,9,7,8]
\]
\[
\text{HEAPIFY( } [1,2,4,6,5,9,7,8], 1 \text{ ) } = [1,2,4,6,5,9,7,8]
\]

Since we make \(O(n)\) calls to \text{HEAPIFY} and since each one takes \(O(\log n)\) time, we immediately get a bound of \(O(n \log n)\). But in fact, we can do better by analyzing more carefully: basically, we call \text{HEAPIFY} on each subtree of height \(\geq 1\) and \text{HEAPIFY} runs in time proportional to the height of that subtree. So we can estimate the total running time as follows:

\[
O(\sum_{h=1}^{\log n} h \times \text{number of subtrees of height } h).
\]

The sum goes to \(\log n\) because the height of the whole tree is \(\Theta(\log n)\). A tree with \(n\) nodes contains at most \(\lceil n/2^{h+1} \rceil\) nodes of height \(h\) (why?), so it contains at most the same number of subtrees of height \(h\). Therefore, the running time is:

\[
O(\sum_{h=1}^{\log n} h \times \lceil n/2^{h+1} \rceil) = O(n \sum_{h=1}^{\infty} h/2^h) = O(n).
\]

The last equation comes from the fact that \(\sum_{h=1}^{\infty} h/2^h \leq 2\) (Page 1061 (A.8) of CLRS). So \text{BUILD-HEAP} runs in time \(O(n)\).

### 18.4 Complexity of Prim’s Algorithm

So what is the running time of Prim’s Algorithm? It turns out that using the above ideas, you can implement \text{DECREASE-PRIORITY} in time \(O(\log n)\). The while loop of Prim’s runs at most once for every vertex in the graph. If \(u\) is the current vertex selected from the priority queue, then the algorithm analyzes each edge going from \(u\) to outside \(A\). Since \(u\) is always added to \(A\) and never becomes the current vertex again, we consider each edge at most once. When we consider an edge, we might decrease the priority of one of its endpoints, which takes time \(O(\log n)\). Therefore, the loop takes time at most \(O(m \log n)\). Since the initial building of the heap can be done faster than this, the worst-case running time is \(O(m \log n)\).
19 Kruskal’s Algorithm for MCST

Kruskal’s algorithm uses a Union-Find ADT. We need to define this before proceeding with the algorithm.

19.1 The Disjoint Set ADT (also called the Union-Find ADT)

Two sets $A$ and $B$ are disjoint if their intersection is empty: $A \cap B = \emptyset$. In other words, if there is no element in both sets, then the sets are disjoint. The following abstract data type, called “Disjoint Set” or “Union-Find,” deals with a group of sets where each set is disjoint from every other set (i.e. they are pairwise disjoint).

**Object:** A collection of nonempty, pairwise disjoint sets: $S_1, \ldots, S_k$. Each set contains a special element called its representative.

**Operations:**

- **MAKE-SET(x):** Takes an element $x$ that is not in any of the current sets, and adds the set $\{x\}$ to the collection. The representative of this new set is $x$.

- **FIND-SET(x):** Given an element $x$, return the representative of the set that contains $x$ (or some NIL if $x$ does not belong to any set).

- **UNION(x,y):** Given two distinct elements $x$ and $y$, let $S_i$ be the set that contains $x$ and $S_j$ be the set that contains $y$. This operation adds the set $S_i \cup S_j$ to the collection and it removes $S_i$ and $S_j$ (since all the sets must be disjoint). It also picks a representative for the new set (how it chooses the representative is implementation dependent). Note: if $x$ and $y$ originally belong to the same set, then UNION(x,y) has no effect.

The Union-Find ADT provides us with an easy method for testing whether an undirected graph is connected:

For all $v$ in $V$ do
  MAKE-SET(v)
For all $(u,v)$ in $E$ do
  UNION(u,v)

Now we can test whether there is a path between $u$ and $v$ by testing FIND-SET(u) = FIND-SET(v).

19.2 Pseudocode for Kruskal

```plaintext
KRUSKAL-MST(G=(V,E),w:E->Z)
    A := {};
    sort edges so $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$;
    for each vertex $v$ in $V$, MAKE-SET(v);
    for $i := 1$ to $m$ do
        (let $(u_i,v_i) = e_i$)
        if FIND-SET(u_i) != FIND-SET(v_i) then
```

52
UNION(u_i,v_i);
A := A U {e_i};
end if
end for
END KRUSKAL-MST

Intuitively, Kruskal’s algorithm grows an MCST $A$ by repeatedly adding the “lightest” edge from $E$ that won’t create a cycle.

19.3 Correctness

We can argue correctness in a similar way to the way we proved correctness for Prim’s algorithm.

**Theorem.** If $G = (V, E)$ is a connected, undirected, weighted graph, $A$ is a subgraph of some MCST $T$ of $G$, and $e$ is any edge of minimum weight which does not create a cycle with $A$, then $A \cup \{e\}$ is a subset of some MCST of $G$.

**Proof.** We use a similar argument as before. If $e$ is part of $T$, then we are finished. If not, then $e$ forms a cycle with $T$. If so, there must be some other edge $e'$ that is in $T$ but not contained in $A$ (because $e$ does not form a cycle with $A$). Also, $e'$ cannot form a cycle with $A$, because otherwise, it would form a cycle with $T$. By assumption, $w(e) \leq w(e')$. Let $T' = T \cup \{e\} - \{e'\}$. Then, as before, $w(T') \leq w(T)$ and $A \cup \{e\} \subseteq T'$.

19.4 Data Structures for Union-Find

1. **Linked lists:** Represent each set by a linked list, where each node is an element. The representative element is the head of the list. Each node contains a pointer back to the head. The head also contains a pointer to the tail. We can implement the operations as follows ($list_x$ is the list containing $x$ and $list_y$ is the list containing $y$):
   - **MAKE-SET(x):** Just create a list of one node containing $x$. Time: $O(1)$.
   - **FIND-SET(x):** Just follow $x$’s pointer back to the head and return the head. Time: $O(1)$.
   - **UNION(x,y):** Append $list_y$ to the end of $list_x$. Since we can find the head of $list_y$ and the tail of $list_x$ in constant time, this takes $O(1)$ time. The representative of this combined list is the head of $list_x$, but the nodes of $list_y$ still point to the head of $list_y$. To update them to point to the head of $list_x$, it takes time $\Theta($ length of $list_y)$.

The worst-case sequence complexity for $m$ of these operations is certainly $O(m^2)$: no list will contain more than $m$ elements since we can’t call MAKE-SET more than $m$ times. The most expensive operation is UNION; if we call this $m$ times on lists of length $m$, it will take time $O(m^2)$. Obviously this an overestimate of the time since we can’t call both MAKE-SET and UNION $m$ times.

We can show, however, that the worst-case sequence complexity of $m$ operations is $\Omega(m^2)$. To do this, we have to give a sequence that will take time $\Omega(m^2)$: start by calling MAKE-SET $m/2 + 1$ times on elements $x_1, x_2, \ldots, x_{m/2+1}$. Now do the loop:
for \( i = 2 \) to \( m/2 \) do
\[ \text{UNION} \ (x_i, \ x_1) \]
This will create a longer and longer list that keeps getting appended to a single element. The execution of the loop takes time \( \Theta(m^2) \).

2. **Linked lists with union-by-weight:** Everything remains the same except we will store the length of each linked list at the head. Whenever we do a \( \text{UNION} \), we will take the shorter list and append it to the longer list. So, \( \text{UNION}(x, y) \) will no longer take \( O(\text{length of list}_y) \), but rather \( O(\min\{\text{length(list}_x), \text{length(list}_y}\}) \). This type of union is called “union-by-weight” (where “weight” just refers to the length of the list).

It might seem like union-by-weight doesn’t make much of a difference, but it greatly affects the worst-case sequence complexity. Consider a sequence of \( m \) operations and let \( n \) be the number of \( \text{MAKE-SET} \) operations in the sequence (so there are never more than \( n \) elements in total). \( \text{UNION} \) is the only expensive operation and it’s expensive because of the number of times we might have to update pointers to the head of the list. For some arbitrary element \( x \), we want to prove an upper bound on the number of times that \( x \)’s head pointer can be updated during the sequence of \( m \) operations. Note that this happens only when \( \text{list}_x \) is unioned with a list that is no shorter (because we update pointers only for the shorter list). This means that each time \( x \)’s back pointer is updated, \( x \)’s new list is at least twice the size of its old list. But the length of \( \text{list}_x \) can double only \( \log n \) times before it has length greater than \( n \) (which it can’t have because there are only \( n \) elements). So we update \( x \)’s head pointer at most \( \log n \) times. Since \( x \) could be any of \( n \) possible elements, we do total of at most \( n \log n \) pointer updates. So the cost for all the \( \text{UNION} \)’s in the sequence is \( O(n \log n) \). The other operations can cost at most \( O(m) \) so the total worst-case sequence complexity is \( O(m + n \log n) \).

3. **Trees:** Represent each set by a tree, where each element points to its parent and the root points back to itself. The representative of a set is the root. Note that the trees are not necessarily binary trees: the number of children of a node can be arbitrarily large (or small).

- **MAKE-SET(\( x \)):** Just create a tree with a single node \( x \). Time: \( O(1) \).
- **FIND-SET(\( x \)):** Follow the parent pointers from \( x \) until you reach the root. Return root. Time: \( \Theta(\text{height of tree}) \).
- **UNION(\( x, y \)):** Let \( \text{root}_x \) be the root of the tree containing \( x, \text{tree}_x \), and let \( \text{root}_y \) be the root of the tree containing \( y, \text{tree}_y \). We can find \( \text{root}_x \) and \( \text{root}_y \) using \( \text{FIND-SET}(x) \) and \( \text{FIND-SET}(y) \). Then make \( \text{root}_y \) a child of \( \text{root}_x \). Since we have to do both \( \text{FIND-SET} \)s, the running time is \( \Theta(\max\{\text{height(tree}_x), \text{height(tree}_y\}) \).
The worst-case sequence complexity for $m$ operations is just like the linked list case, since we can create a tree which is just a list:

\[
\text{for } i = 1 \text{ to } m/4 \text{ do} \\
\quad \text{MAKE-SET}(x_i) \\
\text{for } i = 1 \text{ to } m/4 - 1 \text{ do} \\
\quad \text{UNION}(x_{i+1}, x_i)
\]
Creating this tree takes $m/4$ MAKE-SET operations and $m/4 - 1$ UNION operations. The running time for $m/2 + 1$ FIND-SET operations on $x_1$ now is $m/4(m/2 + 1) = \Theta(m^2)$.

**Exercise.** How do we know there is not a sequence of operations that takes longer than $\Theta(m^2)$?

4. **Trees with union-by-rank:** We improved the performance of the linked-list implementation by using “weight” or “size” information during UNION. We will do the same thing for trees, using “rank” information. The rank of a tree is an integer that will be stored at the root:

- **MAKE-SET(x):** Same as before. Set $\text{rank} = 0$.
- **UNION(x,y):** If $\text{rank}(\text{tree}_x) \geq \text{rank}(\text{tree}_y)$ then make $\text{root}_y$ a child of $\text{root}_x$. Otherwise, make $\text{root}_x$ a child of $\text{root}_y$. The rank of the combined tree is $\text{rank}(\text{tree}_x) + 1$ if $\text{rank}(\text{tree}_x) = \text{rank}(\text{tree}_y)$, and $\max\{\text{rank}(\text{tree}_x), \text{rank}(\text{tree}_y)\}$ otherwise. The running time is still $\Theta(\max\{\text{height}(\text{tree}_x), \text{height}(\text{tree}_y)\})$.
- **FIND-SET(x):** Same as before.
We can prove two things about union-by-rank:

(a) The rank of any tree created by a sequence of these operations is equal to its height.
(b) The rank of any tree created by a sequence of these operations is $O(\log n)$, where $n$ is the number of MAKE-SETS in the sequence.

These two facts imply that the running times of FIND-SET and UNION are $O(\log n)$, so the worst-case sequence complexity of $m$ operations is $O(m \log n)$.

5. **Trees with union-by-rank and path compression:** In addition to doing union-by-rank, there is another way to improve the tree implementation of Union-Find: When performing FIND-SET(x), keep track of the nodes visited on the path from $x$ to root$_x$ (in a stack or queue), and once the root is found, update the parent pointers of each of these nodes to point directly to the root. This at most doubles the running time of the current FIND-SET operation, but it can speed up future FIND-SETs. This technique is called “path compression.”

This is the state-of-the-art data structure for Union-Find. Its worst case sequence complexity is $O(m \log^{*} n)$ (see section 22.4 of the text for a proof). The function $\log^{*} n$ is a very slowly growing function; it is equal to the number of times you need to apply log to $n$ before the answer is less than 1. For example, if $n = 15$, then $3 < \log n < 4$, so $1 < \log \log n < 2$ and $\log \log \log n < 1$. So $\log^{*} n = 3$. Also, if $n = 2^{65536} = 2^{2^{22}}$, then $\log^{*} n = 5$.

19.5 **Complexity of Kruskal’s Algorithm**

Let’s assume that $m$, the number of edges, is at least $n - 1$, where $n$ is the number of vertices, otherwise $G$ is not connected and there is no spanning tree. Sorting the edges can be done in time $O(m \log m)$ using mergesort, for example. Let’s also assume that we implement Union-Find using linked-lists with union-by-weight. We do $n$ MAKE-SETS, at most $2m$ FIND-SETs and at most $m$ UNIONs. The first two take time $O(n)$ and $O(m)$, respectively. The last can take time at most $O(n \log n)$ since in that amount of time we would have built up the set of all vertices. Hence, the running time of Kruskal is $O(m \log m + n + m + n \log n) = O(m \log m)$.
20 Quicksort

Quicksort sorts a list of integers as follows:

Quicksort ( List R )
  
  if |R| \leq 1 then
    return R
  else
    select pivot a in R
    partition R into
    L = elements less than a
    M = elements equal to a
    U = elements greater than a
    return List( Quicksort(L), M, Quicksort(U) )

For now, we’ll select the first element of R as the pivot a.

20.1 Worst-case analysis

We’ll get an upper bound and then a lower bound on $T_{wc}(n)$, the worst-case running time of Quicksort for inputs of size $n$. As in the ListSearch example, we’ll measure running time in terms of the number of comparisons performed. It will turn out that the upper and lower bounds are the same!

20.1.1 Upper bound

Quicksort works by comparing elements of $R$ with the pivot $a$. Each element of $R$ gets to be the pivot at most once, and it then gets compared to elements which have not yet been used as the pivot. So, we never compare the same pair of elements twice. Hence we perform at most 

$\binom{n}{2} = \frac{n(n-1)}{2}$

comparisons.

20.1.2 Lower bound

To get a lower bound, we guess the worst input for Quicksort and observe how many comparisons are needed to sort it. The quantity $T_{wc}(n)$ must, by definition, be at least this number. Let’s let $R$ be the already sorted list $\ell_n = (1, 2, \ldots, n)$. We start by choosing 1 as the pivot, then comparing the rest of the $n - 1$ elements with 1. Everything is greater than 1 so it all ends up in $U = (2, 3, \ldots, n)$. Now we have to run Quicksort on $U$, which is just as bad as $R$ (since it’s already sorted) except that it is smaller by one element. So, if $t(n)$ is the number of comparisons needed for $\ell_n$, then

$$t(n) = n - 1 + t(n - 1)$$

(19)
for all \( n > 1 \), and \( t(1) = 0 \). If we plug in \( n - 1 \) for \( n \), then we get 
\[
t(n - 1) = n - 2 + t(n - 2).
\]
We can substitute this quantity for \( t(n - 1) \) in (19) to get 
\[
t(n) = n - 1 + n - 2 + t(n - 2).
\]
Next we can substitute for \( t(n - 2) \) in terms of \( t(n - 3) \) and continue until we get to \( t(1) \). So,
\[
t(n) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}.
\]
Hence, we perform \( \frac{n(n-1)}{2} \) comparisons for this \( R \).

Since the upper and lower bounds are the same, we know that \( T_{wc}(n) \) must be exactly \( \frac{n(n-1)}{2} \) for quicksort. Therefore, \( T_{wc}(n) \in \Theta(n^2) \).

### 20.2 Average case analysis

Let’s see if Quicksort does better on average than it does in the worst case.

1. Our sample space \( S_n \) will be all the permutations of the list \( (1, 2, \ldots, n) \) since we don’t care what the actual values of the elements are, just how they’re ordered.

   **Exercise.** *Why is it a reasonable assumption that there are no repeated elements?*

2. Our probability distribution will be the uniform one; that is, we’ll assume all permutations are equally likely and therefore have probability \( \frac{1}{n!} \).

3. Let the random variable \( t_n : S_n \rightarrow \mathbb{N} \) be the number of comparisons needed to sort a given list in \( S_n \).

Recall that the definition of \( T_{avg}(n) \), the average case complexity of a list of length \( n \), is
\[
T_{avg}(n) \overset{d}{=} E[t_n] = \sum_{x \in S_n} \text{Pr}(x)t_n(x).
\]  
(20)

We don’t want to have to consider each individual input in order to compute \( T_{avg}(n) \), so let’s group them together into categories. Let \( A_{ni} \subset S_n \) be such that \( A_{ni} = \{ \text{All permutations of } (1, 2, \ldots, n) \text{ such that } i \text{ is the first element} \} \). \( A_{ni} \) occurs with probability \( \frac{1}{n} \) because each element is equally likely to be the first. If this is the case, then elements \( 1, 2, \ldots, i-1 \) go into \( L \) and elements \( i+1, i+2, \ldots, n \) go into \( U \). All the orderings of \( L \) and \( U \) are equally likely since all the orderings of the original list were equally likely. So, if \( T_{avg}(n) \) is the average case complexity of a list of length \( n \), then \( t_{avg}(A_{ni}) \)—the average number of comparisons needed to sort a list in \( A_{ni} \)—is
\[
t_{avg}(A_{ni}) = n - 1 + T_{avg}(i - 1) + T_{avg}(n - i),
\]
where the three terms on the right are for (i) partitioning into \( L \) and \( U \), (ii) sorting \( L \), and (iii) sorting \( U \).

Let’s try to rewrite (20) in terms of \( A_{ni} \)’s. We can partition the sum over \( S_n \) into a sum of sums over each \( A_{ni} \):
\[
T_{avg}(n) = \sum_{i=1}^{n} \sum_{x \in A_{ni}} \text{Pr}(x)t_n(x).
\]
Since $t_{\text{avg}}(A_{ni})$ is the average time that $x \in A_{ni}$ takes, we can write

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} \left( \sum_{x \in A_{ni}} \Pr(x) t_{\text{avg}}(A_{ni}) \right).$$

The sum $\sum_{x \in A_{ni}} \Pr(x)$ is just the definition of $\Pr(A_{ni})$, so

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} \Pr(A_{ni}) t_{\text{avg}}(A_{ni})$$

$$= \sum_{i=1}^{n} \frac{1}{n} (n - 1 + T_{\text{avg}}(i - 1) + T_{\text{avg}}(n - i))$$

$$= n - 1 + \frac{2}{n} \sum_{j=1}^{n-1} T_{\text{avg}}(j)$$

In addition, we know that $T_{\text{avg}}(0) = T_{\text{avg}}(1) = 0$.

This seems like a difficult recurrence to solve, but observe the similarities between the following two equations:

$$T_{\text{avg}}(n) = n - 1 + \frac{2}{n} \sum_{j=1}^{n-1} T_{\text{avg}}(j) \quad (21)$$

$$T_{\text{avg}}(n - 1) = n - 2 + \frac{2}{n - 1} \sum_{j=1}^{n-2} T_{\text{avg}}(j). \quad (22)$$

Our method will be to eliminate the denominators from the summations and subtract. We get the following equation:

$$nT_{\text{avg}}(n) - (n - 1)T_{\text{avg}}(n - 1) = n(n - 1) - (n - 1)(n - 2) + 2T_{\text{avg}}(n - 1).$$

Then, we simplify:

$$nT_{\text{avg}}(n) = (n + 1)T_{\text{avg}}(n - 1) + 2(n - 1)$$

or

$$\frac{T_{\text{avg}}(n)}{n + 1} = \frac{T_{\text{avg}}(n - 1)}{n} + \frac{2(n - 1)}{n(n + 1)}.$$

If we let $B(n) = \frac{T_{\text{avg}}(n)}{n + 1}$ then

$$B(n) = B(n - 1) + \frac{2(n - 1)}{n(n + 1)},$$

where $B(0) = T_{\text{avg}}(0)/1 = 0$. Then,
\[ B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \]  
\[ = 2 \sum_{i=1}^{n} \frac{1}{i+1} - 2 \sum_{i=1}^{n} \frac{1}{i(i+1)} \]  
\[ = 2 \sum_{i=1}^{n+1} \frac{1}{i} - 2 \sum_{i=1}^{n} \frac{1}{i(i+1)}. \]  

[CLRS Appendix A] shows that \( \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n) \). Clearly, the first term of \( B(n) \) dominates the second and third terms, so \( B(n) \in \Theta(\log(n+1)) = \Theta(\log n) \). Since \( T_{avg}(n) = (n+1)B(n) \), \( T_{avg}(n) \in \Theta(n \log n) \).

If you solve the recurrence more carefully, you will find that all of the constants which are eliminated by the \( \Theta \) notation are small values. This fact is one of the reasons that Quicksort is actually quick in practise (compared with other sorting algorithms that have complexity \( \Theta(n \log n) \)).

### 20.3 Randomized Quicksort

In this section we will be discussing a randomized version of Quicksort. Although randomized Quicksort may appear related to the average case analysis of non-randomized Quicksort, they are not the same. You should be certain that you understand the previous sections on average case analysis before proceeding.

We have seen that Quicksort does well on the average input, but we have also seen that there are some particular inputs on which it does badly. If our input is typically sorted or close to sorted then Quicksort is not a good solution.

One way to fix this situation is to pick a random element as the pivot instead of the first element. We’ll call this algorithm RQuicksort. Note that this is a different algorithm from Quicksort; Quicksort is deterministic, while RQuicksort is randomized. In other words, where Quicksort always chooses the same element as the pivot, RQuicksort chooses a random element as the pivot.

For a given input \( R \), we start by picking a random index \( p_1 \) as the pivot. Then, when we recurse on \( L \) and \( U \), we pick random indices \( p_2 \) and \( p_3 \), respectively, as the pivots, etc. Let \( p = (p_1, p_2, \ldots, p_n) \) be the sequence of random pivot choices in one execution of RQuicksort on a particular input list \( R \). The possible \( p \)'s constitute a sample space \( P_n \), and we’ll assume that the probability distribution on the space is uniform. We can then define the random variable \( t_R : P_n \to \mathbb{N} \), the running time of RQuicksort on list \( R \) given some sequence of pivot choices. The expected running time of RQuicksort on input \( R \) is defined as

\[ E[t_R] = \sum_{p \in P_n} \Pr(p) t_R(p). \]

Note that \( E[t_R] \) is the expected running time for RQuicksort on a given input whereas \( T_{avg}(n) \) is the expected running time for an algorithm over all possible inputs.

Despite this fact, in this case, \( T_{avg}(n) \) and \( E[t_R] \) for any input \( R \) happen to have the same value. This is because choosing a random element of \( R \) is equivalent to choosing the first element of a random permutation of \( (1, 2, \ldots, n) \). So \( E[t_R] = \Theta(n \log n) \) for any \( R \). This is good because there is no particular input which will definitely be bad for RQuicksort.
In general, the expected running time of a randomized algorithm $A$ may vary depending upon the input. As usual, let $S_n$ be the possible inputs to $A$ of size $n$. Let $P_n(x)$ be the sample space of random choices that $A$ can make on input $x$. We can define the expected worst-case complexity of $A$ as

$$\max_{x \in S_n} \{E[t_x]\},$$

where the expectation is over $P_n(x)$.

Instead of relying on unknown distribution of inputs, randomize an algorithm by picking random element as pivot. This way, random behaviour of an algorithm on any fixed input is equivalent to fixed behaviour of the same algorithm on a uniformly random input. In other words, expected worst case complexity of RQuicksort is $\Theta(n \log(n))$. In general, randomized algorithms are good when there are many good choices but it is difficult to find one choice that is guaranteed to be good.
21 Lower bounds

Definition. A comparison-based algorithm is an algorithm where the behaviour of the algorithm is based only on the comparisons between elements.

In a comparison-based sort, we only use comparisons between elements to gain information about the order of a sequence. Given two elements $a_i$ and $a_j$, we perform one of the tests: $a_i < a_j$, $a_i = a_j$, or $a_i > a_j$. We can say that each test returns one of the following outcomes: $(\leq, >)$, $(<, \geq)$, $(\leq, \geq)$, $(<, =, >)$, $(=, \neq)$

We can express a comparison sort as a decision tree.

Example

Let’s look at a particular decision tree for sorting the set \{A, B, C\}. Internal nodes represent comparisons of the two elements in the node. Leaf nodes represent the result of the sort. Each edge is labelled with the outcome of the comparison of the node above it.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{decision_tree.png}
\caption{Decision tree for sorting \{A, B, C\}.}
\end{figure}

Note that this is only a particular decision tree for this sort. There are other possibilities. The decision tree depends on the algorithm that we are using. Observe that this decision tree has 6 leaves and every permutation of the elements occurs as some leaf node.

The length of the longest path from the root of the decision tree to a leaf represents the worst-case number of comparisons that the sorting algorithm performs. Therefore, worst-case number of comparisons is the height of the decision tree.

How can we find the algorithm with the smallest worst-case number of comparisons? We find the decision tree with the smallest height.
In other words, to find the worst-case running time of the "best" algorithm (i.e., the one with the smallest worst-case number of comparisons), we want to find a lower bound on the height of the decision trees.

Some useful facts:

1. The number of ways to sort \( n \) numbers is \( n! \) (each permutation of the numbers is possible). This implies that the number of leaves in the decision tree is at least \( n! \) (there may be duplicate leaves).

2. There are at most 3 outcomes (branches) at any level of the tree: (\(<\), \(=\), \(>\)). Recall the other possible outcomes are: (\(\leq, >\)), (\(<, \geq\)), (\(=, \neq\)).

3. A ternary tree of height \( h \) has at most \( 3^h \) leaves. Alternatively, a tree with at least \( 3^h \) leaves must have height at least \( h \).

Thus, we can conclude that the decision tree for sorting has height at least \( \log_3 n! \).

Since \( \log_3 n! \in \Theta(n \log n) \), then \( h \in \Omega(n \log n) \). This implies that the worst-case number of comparisons for any comparison-based sorting algorithm is in \( \Omega(n \log n) \).

### 21.1 Showing that \( \log_3 n! \in \Theta(n \log n) \)

First, recall that \( \log_2 n! \in \Theta(n \log n) \) due to Stirling's Approximation. It is then easy to show that \( \log_3 n! \in \Theta(n \log_3 n) \).

Then,

\[
\begin{align*}
    k &= \log_3 f(n) \\
    3^k &= f(n) \quad \text{(take both sides as the exponent of 3)} \\
    k \log_2 3 &= \log_2 f(n) \quad \text{(take the \log_2 of both sides)} \\
    \log_3 f(n) &= \log_2 f(n) / \log_2 3 \quad \text{(replace \( k \) with \( \log_3 f(n) \), and divide both sides by \( \log_2 3 \))}
\end{align*}
\]

Therefore \( \log_3 n! \in \Theta(\log_2 n!) \) and \( \log_3 n! \in \Theta(n \log n) \).

### 21.2 Another example

Let’s find a lower bound on the worst-case time needed to compute the following problem:

Is there an element that appears exactly 5 times and another element that appears exactly 7 times in an \( n \)-element array of integers?

Consider the following algorithm:

1. **Step 1** Sort all the elements.
2. **Step 2** Scan through the elements and count whether one element appears 5 times and another appears 7 times.
The total worst-case time of this algorithm is in \( O(n \log n) \) So, either the lower bound on the worst-case time needed is \( \Omega(n \log n) \) or it is something smaller.

Let's compute the lower bound using the decision tree method. Consider the following:

1. There are two possible solutions to this problem: Yes and No. This implies that the number of leaves in the decision tree is at least 2 (there may be duplicate leaves).
2. There are at most 3 outcomes (branches) at any level of the tree: \(<, =, >\)

We can conclude that the decision tree for sorting has height at least \( \log_3 2 \) Since \( \log_3 2 \) in \( \Omega(1) \), we have a lower bound of \( \Omega(1) \).

But, our lower and upper bounds don't match, and there doesn't seem to be a constant time algorithm that solves this problem. What do we do?

### 21.2.1 The adversary method

This method pits our algorithm against an adversary. Our algorithm asks a question and then tries to reduce choices as quickly as possible (i.e. tries to learn the input). The adversary answers questions such that at least one input is consistent with the answers and tries to keep the legal set of inputs as big as possible (i.e. it tries to make you do as much work as possible).

**Example**

The game "Guess Who?" This is a game where you try to determine a particular person by asking questions about them. I always answer correctly but keep the answer so that as many people as possible are still available (because I haven't chosen the answer yet). You are our algorithm and I am the adversary.

We will use an adversary argument to solve the problem above. Suppose the algorithm only take \( n-1 \) steps. Then, it only reads \( n-1 \) spots in the array. The adversary let the algorithm read the following input: \( x x x x x y y y y y y y \ldots \). The algorithm is correct, and saw 5 \( x \)'s and 7 \( y \)'s so it answers 'yes', but it did not see one of the input values.

We have the following cases:

1. The algorithm did not see a '.' value If the adversary changes the input so the '.' becomes an 'x', we have: \( x x x x x y y y y y y y \ldots x \). The algorithm behaves the same way, and answers incorrectly!
2. The algorithm did not see a 'x'.
3. The algorithm did not see a 'y'.

For case each, we can change the input so that the algorithm answers incorrectly!

Therefore, we conclude that the algorithm needs to take at least \( n \) steps. This implies that a lower bound for the worst-case time for ANY algorithm to solve this problem is in \( \Omega(n) \). This is better than the \( \Omega(1) \) lower bound from the decision tree method.

**Exercise.** Can we find an algorithm that takes linear time or is our lower bound still too low?
21.3 Lower bounds for comparison based search

21.3.1 Searching a sorted array

Example
The algorithm for binary search is as follows (where we return null when the key k is not found):

\[
\text{BinarySearch}(A, k) \\
\text{low} = 1 \\
\text{high} = \text{size}(A) \\
\text{repeat until (high < low)} \\
\quad \text{mid} = \text{floor}((\text{high}-\text{low})/2) + \text{low} \\
\quad \text{if } A[\text{mid}] = k \\
\quad \quad \text{return mid} \\
\quad \text{else if } A[\text{mid}] < k \\
\quad \quad \text{low} = \text{mid} + 1 \\
\quad \text{else if } A[\text{mid}] > k \\
\quad \quad \text{high} = \text{mid} - 1 \\
\text{return null}
\]

The decision tree is as follows:

Some observations:
1. Some leaves are duplicates (i.e. the ones that return null).
2. There are \( n + 1 \) unique leaves (we can return each of the 5 keys, and null if the value is not in the array).
3. Each non-leaf node has three children.

For any comparison-based search algorithm A, we can prove a lower bound of $\Omega(\log n)$ as follows:

- The number of distinct outcomes is $n+1$ in the worst-case (i.e. when all values in the array are unique).
- The corresponding decision tree $T_A$ has at least $n + 1$ leafs.
- There are at most three outcomes at any tree level ($=, <, >$).
- The decision tree has height at least $\log_3 n$ (since a ternary tree of height $h$ has at most $3^h$ leaves).
- Since $\log_3 n \in \Omega(\log n)$, we have a lower bound of $\Omega(\log n)$.

### 21.3.2 Searching an unsorted array

Sometimes the decision tree technique to determine a lower bound is not powerful enough to give a good lower bound. In particular, consider any algorithm for searching on an unsorted array.

- The decision tree has $n + 1$ distinct leafs
- The tree has 2 outcomes at each level ($=, \neq$).
- The tree has height $\log_2 n$.
- We have a lower bound of $\Omega(\log n)$.

But, our lower bound is too low! Search on an unsorted array actually has a lower bound of $\Omega(n)$. Intuitively, if we only look at $n - 1$ elements, we can’t be sure if an element is not in the array or if it is the element we haven’t looked at.

**Exercise.** Why is there a discrepancy between the decision tree lower bound and our intuitive understanding of the lower bound?