1 Intro

Lecture 1, 1/16/2020

- Handout 1 — My names
- TA: Patrick Shepherd
- Plagiarism — read aloud
- Assignments on web. Use C, C++, or Java.
- E-mail list: CS350001@cs.uky.edu
- accounts in Multilab
- text — we will skip around

2 Basic building blocks: Linked lists (Chapter 3) and trees (Chapter 4)

Linked lists and trees are examples of data structures:

- way to represent information
- so it can be manipulated
- packaged with routines that do the manipulations

Leads to an Abstract Data Type (ADT): has an API (specification) and hides its internals.
3 Tools

<table>
<thead>
<tr>
<th>Use</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td></td>
</tr>
</tbody>
</table>

4 Linked list

- used as a part of several ADTs.
- Can be considered an ADT itself.
- Collection of nodes, each with optional arbitrary data and a pointer to the next element on the list.

![Linked list diagram]

**operation**  | **cost**  
create empty list | $O(1)$  
insert new node at front of list | $O(1)$  
delete first node, returning data | $O(1)$  
count length | $O(n)$  
search by data | $O(n)$  
sort | $O(n \log n)$ to $O(n^2)$
5 Sample code

```c
#define NULL 0
#include <stdlib.h>

typedef struct node_s {
    int data;
    struct node_s *next;
} node;

node *makeNode(int data, node* next) {
    node *answer = (node *) malloc(sizeof(node));
    answer->data = data;
    answer->next = next;
    return (answer);
} // makeNode

node *insertHead(node* handle, int data) {
    return makeNode(data, handle->next);
} // insertHead

node *searchDataI(node *handle, int data) {
    // iterative method
    node *current = handle->next;
    while (current != NULL) {
        if (current->data == data) break;
        current = current->next;
    }
    return current;
} // searchDataI

node *searchDataR(node *handle, int data) {
    // recursive method
    node *current = handle->next;
    if (current == NULL) return NULL;
    else if (current->data == data) return current;
    else return searchDataR(current, data);
} // searchDataR
```
6 Improving the efficiency of some operations

- To make insert at end fast: maintain two handles, one to the front, the other to the rear of the list.
- To make count() fast: maintain the count in a separate variable. If we need the count more often than we insert and delete, it is worthwhile.
- Combine these new items in a header node:

```c
typedef struct {
    node *front;
    node *rear;
    int count;
} nodeHeader;
```

- To make search faster: remove the special case that we reach the end of the list by placing a pseudo-data node at the end. Keep track of the pseudo-data node in the header.

```c
typedef struct {
    node *front;
    node *rear;
    node *pseudo;
    int count;
} nodeHeader;
```

```c
node *searchDataI(nodeHeader *header, int data) {
    // iterative method
    header->pseudo->data = data;
    node *current = header->front;
    while (current->data != data) {
        current = current->next;
    }
    return (current == header->pseudo ? NULL : current);
} // searchDataI
```

- Exercise: If we want both pseudo-data and a rear pointer, how does an empty list look?
- Exercise: If we want pseudo-data, how does searchDataR() change?
- Exercise: Is it easy to add a new node after a given node?
Exercise: Is it easy to add a new node before a given node?

7 Queues, stacks, dequeues: built out of either linked lists or arrays

- We’ll see each of these.

8 Stack of integer

- Abstract definition: either empty or the result of pushing an integer onto the stack.
- operations
  - stack makeEmptyStack()
  - boolean isEmptyStack(stack S)
  - int popStack(stack *S) // modifies S
  - void pushStack(stack *S, int I) // modifies S

9 Implementation 1 of Stack: Linked list

- makeEmptyStack implemented by makeEmptyList()
- isEmptyStack implemented by isEmptyList()
- pushStack inserts at the front of the list
- popStack deletes from the front of the list

10 Implementation 2 of Stack: Array

- Warning: it’s easy to make off-by-one errors.
The array implementation limits the size. Does the linked-list implementation also limit the size?

The array implementation needs one cell for (potential) element, and one for the count. How much space does the linked-list implementation need?

We can position two opposite-sense stacks in one array so long as their combined size never exceeds MAXSTACKSIZE.
11 Queue of integer

- Abstract definition: either empty or the result of inserting an integer at the rear of a queue or deleting an integer from the front of a queue.
- operations
  - queue makeEmptyQueue()
  - boolean isEmptyQueue(queue Q)
  - void insertQueue(queue Q, int I) // modifies Q
  - int deleteQueue(queue Q) // modifies Q

12 Aside: Unix pipes

- Unix programs automatically have three “files” open: standard input, which is by default the keyboard, standard output, which is by default the screen, and standard error, which is by default the screen.
- In C and C++, they are defined in stdio.h by the names stdin, stdout, and stderr.
- The command interpreter (in Unix, it’s called the “shell”) lets you invoke programs redirecting any or all of these three. For instance, `ls | wc` redirects stdout of the `ls` program to stdin of the `wc` program.
- If you run your `cards` program without redirection, you can type in arbitrary numbers.
- If you run `randGen.pl` without redirection, it generates an unbounded list of pseudo-random numbers to stdout.
- If you run `randGen.pl | cards`, the list of numbers from `randGen.pl` is redirected as input to `cards`.

13 Implementation 1 of Queue: Linked list

We use a header to represent the front and the rear, and we put a dummy node at the front.
```c
#include <stdlib.h>

typedef struct node_s {
    int data;
    struct node_s *next;
} node;

typedef struct {
    node *front;
    node *rear;
} queue;

queue *makeEmptyQueue() {
    queue *answer = (queue *) malloc(sizeof(queue));
    answer->front = answer->rear = makeNode(0, NULL);
    return answer;
} // makeEmptyQueue

bool isEmptyQueue(queue *theQueue) {
    return (theQueue->front == theQueue->rear);
} // isEmptyQueue

void insertQueue(queue *theQueue, int data) {
    node *newNode = makeNode(data, NULL);
    theQueue->rear->next = newNode;
    theQueue->rear = newNode;
} // insertQueue

int deleteQueue(queue *theQueue) {
    if (isEmptyQueue(theQueue)) return error("queue underflow");
    node *oldNode = theQueue->front->next;
    theQueue->front->next = oldNode->next;
    if (theQueue->front->next == NULL) {
        theQueue->rear = theQueue->front;
    }
    return oldNode->data;
} // deleteQueue
```
14 Implementation 2 of Queue: Array

Warning: it’s easy to make off-by-one errors.
#define MAXQUEUESIZE 30

typedef struct {
    int contents[MAXQUEUESIZE];
    int front; // index of element at the front
    int rear; // index of first free space after the queue
} queue;

bool isEmptyQueue(queue *theQueue) {
    return (theQueue->front == theQueue->rear);
} // isEmptyQueue

int advance(int index) { // circular advance
    return (index + 1) % MAXQUEUESIZE;
} // advance

void insertQueue(queue *theQueue, int data) {
    if (advance(theQueue->rear) == theQueue->front)
        error("queue overflow");
    else {
        theQueue->contents[theQueue->rear] = data;
        theQueue->rear = advance(theQueue->rear);
    }
} // insertQueue

int deleteQueue(queue *theQueue) {
    if (isEmptyQueue(theQueue)) {
        return error("queue underflow");
    } else {
        int answer = theQueue->contents[theQueue->front];
        theQueue->front = advance(theQueue->front);
        return answer;
    }
} // deleteQueue
15 Dequeue of integer

- Abstract definition: either empty or the result of inserting an integer at the front or rear of a dequeue or deleting an integer from the front or rear of a queue.

- Operations
  - dequeue makeEmptyDequeue()
  - boolean isEmptyDequeue(dequeue D)
  - void insertFrontDequeue(dequeue D, int I) // modifies D
  - void insertRearDequeue(dequeue D, int I) // modifies D
  - int deleteFrontDequeue(dequeue D) // modifies D
  - int deleteRearDequeue(dequeue D) // modifies D

- Exercise: code the insertFrontDequeue() and deleteRearDequeue() routines using an array.

- A singly-linked list is fine except for deleteRearDequeue(), which becomes $O(n)$.

- The proper list structure is a doubly-linked list with a single dummy.

- Exercise: Code all the routines.

- Exercise: Is it easy to add a new node after a given node?

- Exercise: Is it easy to add a new node before a given node?
16 Searching

- Given \( n \) data elements (we will use integer data), arrange them in a data structure \( D \) so that these operations are fast:
  - insert(int data, *D)
  - boolean search(int data, D) (can also return entire data record)

- We don’t care about the speed of deletion (for now).
- Much of this material is in Chapter 4 of the book (trees)

- Representation 1: Linked list
  - insert(i) is \( \mathcal{O}(1) \) (place at front)
  - search(i) is \( \mathcal{O}(n) \) (may need to look at whole list; use pseudo-data \( i \) to make search as fast as possible)

- Representation 2: Sorted linked list
  - insert(i) is \( \mathcal{O}(n) \) (average: \( n/2 \) steps)
  - search(i) is \( \mathcal{O}(n) \) (may need to look at whole list; on average, look at \( n/2 \) elements. Use pseudo-data (value \( \infty \)) to make search as fast as possible.)

- Lecture 4, 1/28/2020

- Representation 3: Array
  - insert(i) is \( \mathcal{O}(1) \) (place at end)
  - search(i) is \( \mathcal{O}(n) \) (may need to look at whole list; use pseudo-data \( i \) at end)

- Representation 4: Sorted array
  - insert(i) is \( \mathcal{O}(n) \) (need to search, then shove members over)
• search(i) is $O(\log n)$ (binary search)

```cpp
bool search(int target, int *array, int lowIndex, int highIndex) {
    // look for target in array[lowIndex..highIndex]
    while (lowIndex < highIndex) { // at least 2 elements
        int mid = (lowIndex + highIndex) / 2; // round down
        if (array[mid] < target) lowIndex = mid + 1;
        else highIndex = mid;
    } // while at least 2 elements
    return (array[lowIndex] == target);
} // search
```
17 Quadratic search: set mid based on discrepancy

Also called interpolation search, extrapolation search, dictionary search.

```java
bool search(int target, int *array, int lowIndex, int highIndex) {
    // look for target in array[lowIndex..highIndex]
    while (lowIndex < highIndex) {
        // at least 2 elements
        if (array[highIndex] == array[lowIndex]) {
            highIndex = lowIndex;
            break;
        }
        float percent = (0.0 + target - array[lowIndex])
            / (array[highIndex] - array[lowIndex]);
        int mid = int(percent * (highIndex-lowIndex)) + lowIndex;
        if (mid == highIndex) {
            mid -= 1;
        }
        if (array[mid] < target) {
            lowIndex = mid + 1;
        } else {
            highIndex = mid;
        }
    } // while at least 2 elements
    return (array[lowIndex] == target); // search
}
```

Experimental results

- It is hard to program correctly.
- For $10^6 \approx 2^{20}$ elements, binary search always makes 20 probes.
- This result is consistent with $O(\log n)$.
- Quadratic search: 20 tests with uniform data. The range of probes was 3 – 17; the average about 9 probes.
- Analysis shows that if the data are uniformly distributed, quadratic search should be $O(\log \log n)$. 
18 Analyzing binary search

• Binary search: \( c_n = 1 + c_{n/2} \) where \( c_n \) is the number of steps to search for an element in an array of length \( n \).

• We will use the **Recursion Theorem**: if \( c_n = f(n) + ac_n/b \), where \( f(n) = \Theta(n^k) \), then

\[
\begin{array}{c|c}
\text{when} & \text{when} \\
 a < b^k & c_n \equiv \Theta(n^k) \\
 a = b^k & c_n \equiv \Theta(n^k \log n) \\
 a > b^k & c_n \equiv \Theta(n^{\log a})
\end{array}
\]

• In our case, \( a = 1 \), \( b = 2 \), \( k = 0 \), so \( b^k = 1 \), so \( a = b^k \), so \( c_n = \Theta(n^k \log n) = \Theta(\log n) \).

• Bad news: any comparison-based searching algorithm is \( \Omega(\log n) \), that is, needs at least on the order of \( \log n \) steps.

• Notation, slightly more formally defined. All these ignore multiplicative constants.
  - \( O(f(n)) \): no worse than \( f(n) \); at most \( f(n) \).
  - \( \Omega(f(n)) \): no better than \( f(n) \); at least \( f(n) \).
  - \( \Theta(f(n)) \): no better or worse than \( f(n) \); exactly \( f(n) \).

19 Representation 5: Binary tree

• Example with elicited values

• Pseudo-data: in the universal “null” node.

• \( \text{insert}(i) \) and \( \text{search}(i) \) are both \( O(\log n) \) if we are lucky or data are random.
#define NULL 0
#include <stdlib.h>

typedef struct treeNode_s {
    int data;
    treeNode_s *left, *right;
} treeNode;

treeNode *makeNode(int data) {
    treeNode *answer = (treeNode *) malloc(sizeof(treeNode));
    answer->data = data;
    answer->left = answer->right = NULL;
    return answer;
} // makeNode

treeNode *searchTree(treeNode *tree, int key) {
    if (tree == NULL) return (NULL);
    else if (tree->data == key) return (tree);
    else if (key < tree->data)
        return (searchTree(tree->left, key));
    else
        return (searchTree(tree->right, key));
} // searchTree

void insertTree(treeNode *tree, int key) {
    // assumes empty tree is a pseudo-node with infinite data
    treeNode *parent = NULL;
    treeNode *newNode = makeNode(key);
    while (tree != NULL) {
        parent = tree;
        tree = (key <= tree->data) ? tree->left : tree->right;
    } // dive down tree
    if (key <= parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;
} // insertTree

• We will deal with balancing trees later.
20 Traversals

- A traversal walks through the tree, visiting every node.
- **Symmetric traversal** (also called **inorder**)

```c
void symmetric(treeNode *tree) {
    if (tree == NULL) { // do nothing
    } else {
        symmetric(tree->left);
        visit(tree);
        symmetric(tree->right);
    }
} // symmetric()
```

- **Pre-order traversal**

```c
void preorder(treeNode *tree) {
    if (tree == NULL) { // do nothing
    } else {
        visit(tree);
        preorder(tree->left);
        preorder(tree->right);
    }
} // preorder()
```

- **Post-order traversal**

```c
void postorder(treeNode *tree) {
    if (tree == NULL) { // do nothing
    } else {
        postorder(tree->left);
        postorder(tree->right);
        visit(tree);
    }
} // postorder()
```

21 Representation 6: Hashing (scatter storage)

- Hashing is often the best method.
- insert(data) and search(data) are $\mathcal{O}(\log n)$, but we can generally treat them as $\mathcal{O}(1)$.  


• We will discuss hashing later.

22 Finding the $j$th largest element in a set

• Lecture 6, 2/4/2020

• If $j = 1$, a single pass works in $O(n)$ time:

```cpp
largest = -∞; // priming
foreach (value in set) {
  if (value > largest) largest = value;
}
```

• If $j = 2$, a single pass still works in $O(n)$ time, but it is about twice as costly:

```cpp
largest = nextLargest = -∞; // priming
foreach (value in set) {
  if (value > largest) {
    nextLargest = largest;
    largest = value;
  } else if (value > nextLargest) {
    nextLargest = value;
  }
}
```

• It appears that for arbitrary $j$ we need $O(jn)$ time, because each iteration needs $t$ tests, where $1 ≤ t ≤ j$, followed by sliding $j + 1 - t$ values over, for a total cost of $j + 1$.

• Clever algorithm using an array: QuickSelect (Tony Hoare)
  • Partition the array into “small” and “large” elements with a pivot between them (details soon).
  • Recurse in either the small or large subarray, depending where the $j$th element falls. Stop if the $j$th element is the pivot.

• Cost: $n + n/2 + n/4 + \ldots = 2n = O(n)$

• We can also compute the cost using the recursion theorem (page 16):
  • $c_n = n + c_{n/2}$ (if we are lucky)
  • $c_n = n + c_{2n/3}$ (fairly average case)
• \( f(n) = n = O(n^1) \)
• \( k = 1, a = 1, b = 2 \) (or \( b = 3/2 \))
• \( a < b^k \)
• so \( c_n = \Theta(n^k) = \Theta(n) \)

23 Partitioning an array

• Nico Lomuto’s method

• Partitions array[lowIndex .. highIndex] into three pieces:
  • array[lowIndex .. divideIndex -1]
  • array[divideIndex]
  • array[divideIndex + 1 .. highIndex]

  The elements of each piece are in order with respect to adjacent pieces.

```java
int partition(int array[], int lowIndex, int highIndex) {
    // modifies array, returns pivot index.
    int pivotValue = array[lowIndex];
    int divideIndex = lowIndex;
    for (int combIndex = lowIndex+1; combIndex <= highIndex; combIndex += 1) {
        // array[lowIndex] is the partition (pivotValue) value.
        // array[lowIndex+1 .. divideIndex] are < pivot
        // array[divideIndex+1 .. combIndex-1] are >= pivot
        // array[combIndex .. highIndex] are unseen
        if (array[combIndex] < pivotValue) {
            // see a small value
            divideIndex += 1;
            swap(array, divideIndex, combIndex);
        }
    } // each combIndex
    // swap pivotValue into its place
    swap(array, divideIndex, lowIndex);
    return (divideIndex);
} // partition
```

• Example
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>4</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>5</th>
<th>9</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
24 Using partitioning to select $j$th smallest

Lecture 7, 2/6/2020

```java
void selectJthSmallest (int array[], int size, int targetIndex) {
    // rearrange the values in array[0..size-1] so that
    // array[targetIndex] has the value it would have if the array
    // were sorted.
    int lowIndex = 0;
    int highIndex = size-1;
    while (lowIndex < highIndex) {
        int midIndex = partition(array, lowIndex, highIndex);
        if (midIndex == targetIndex) {
            return;
        } else if (midIndex < targetIndex) { // look to right
            lowIndex = midIndex + 1;
        } else { // look to left
            highIndex = midIndex - 1;
        }
    } // while lowIndex < highIndex
} // selectJthSmallest
```

25 Sorting

- We usually are interested in sorting an array in place.
- Sorting is $\Omega(n \log n)$.
- Good methods are $O(n \log n)$.
- Bad methods are $O(n^2)$.

26 Sorting out sorting

- https://www.youtube.com/watch?v=YvTW7341kpA
- https://www.youtube.com/watch?v=p1Ai7kcqMNU
- https://www.youtube.com/watch?v=gtdfW3TbeYY
- https://www.youtube.com/watch?v=wdcoRfS8edM
27 Insertion sort

- Comb method:

```
sorted    unsorted

probe
```

- \( n \) iterations.
- Iteration \( i \) may need to shift the probe value \( i \) places.
- \( \Rightarrow O(n^2) \).
- Experimental results for Insertion Sort:
  compares + moves \( \approx n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>compares</th>
<th>moves</th>
<th>( n^2 / 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2644</td>
<td>2545</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>9733</td>
<td>9534</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>41157</td>
<td>40758</td>
<td>80000</td>
</tr>
</tbody>
</table>
void insertionSort(int array[], int length) {
    // array goes from 1..length.
    // location 0 is available for pseudo-data.
    int combIndex, combValue, sortedIndex;
    for (combIndex = 2; combIndex <= length; combIndex += 1) {
        // array[1 .. combIndex-1] is sorted.
        // Place array[combIndex] in order.
        combValue = array[combIndex];
        sortedIndex = combIndex - 1;
        array[0] = combValue - 1; // pseudo-data
        while (combValue < array[sortedIndex]) {
            array[sortedIndex+1] = array[sortedIndex];
            sortedIndex -= 1;
        }
        array[sortedIndex+1] = combValue;
    } // for combIndex
} // insertionSort

28 Selection sort

- Comb method:

```
+----------------+----------------+
| sorted, small  | unsorted, large |
+----------------+----------------+
  smallest       
  +----------------+----------------+
  | sorted, small  | unsorted, large |
  +----------------+----------------+
```

- $n$ iterations.
- Iteration $i$ may need to search through $n - i$ places.
- $\Rightarrow O(n^2)$.
- Experimental results for Selection Sort: compares + moves $\approx n$. 
<table>
<thead>
<tr>
<th>n</th>
<th>compares</th>
<th>moves</th>
<th>( n^2/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4950</td>
<td>198</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>19900</td>
<td>398</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>79800</td>
<td>798</td>
<td>80000</td>
</tr>
</tbody>
</table>

```java
void selectionSort(int array[], int length) {
    // array goes from 0..length-1
    int combIndex, smallestValue, bestIndex, probeIndex;
    for (combIndex = 0; combIndex < length; combIndex += 1) {
        // array[0 .. combIndex-1] has lowest elements, sorted.
        // Find smallest other element to place at combIndex.
        smallestValue = array[combIndex];
        bestIndex = combIndex;
        for (probeIndex = combIndex+1; probeIndex < length;
            probeIndex += 1) {
            if (array[probeIndex] < smallestValue) {
                smallestValue = array[probeIndex];
                bestIndex = probeIndex;
            }
        }
        swap(array, combIndex, bestIndex);
    } // for combIndex
} // selectionSort
```
29 Quicksort (C. A. R. Hoare)

- Recursive based on partitioning:

```
  random

  partition

  small  big

  sort   sort
```

- about log $n$ depth.
- each depth takes about $\mathcal{O}(n)$ work.
- $\Rightarrow \mathcal{O}(n \log n)$.
- Can be unlucky: $\mathcal{O}(n^2)$.
- To prevent worst-case behavior, partition based on median of 3 or 5.
- Don’t quicksort regions smaller than about 6 cells; use a final insertionSort pass instead.

- Experimental results for QuickSort:
  compares + moves $\approx 2.1n \log n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>compares</th>
<th>moves</th>
<th>$n \log n$</th>
<th>$n^2/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>643</td>
<td>824</td>
<td>664</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>1444</td>
<td>1668</td>
<td>1528</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>3885</td>
<td>4228</td>
<td>3457</td>
<td>80000</td>
</tr>
<tr>
<td>800</td>
<td>8066</td>
<td>8966</td>
<td>7715</td>
<td>320000</td>
</tr>
<tr>
<td>1600</td>
<td>17583</td>
<td>18958</td>
<td>17030</td>
<td>1280000</td>
</tr>
</tbody>
</table>
```c
void quickSort(int array[], int lowIndex, int highIndex) {
    if (highIndex - lowIndex <= 0) return;
    // could stop if <= 6 and finish by using insertion sort.
    int midIndex = partition(array, lowIndex, highIndex);
    quickSort(array, lowIndex, midIndex-1);
    quickSort(array, midIndex+1, highIndex);
} // quickSort
```

### 30 Shell Sort (Donald Shell, 1959)

- Donald Shell, 1959
- Each pass has a span $s$.

```c
for (int span in reverse(spanSequence)) {
    for (int offset = 0; offset < span; offset += 1) {
        insertionSort(a[offset], a[offset+span], ...)
    }
} // each offset
```  

- The last element in spanSequence must be 1.
- Tokuda’s sequence: $s_0 = 1$; $s_k = 2.25s_{k-1} + 1$; $span_k = \lceil s_k \rceil = 1, 4, 9, 20, 46, 103, 233, 525, 1182, 2660, ...$
- Experimental results for Shell Sort: compares + moves $\approx 2.2n \log n$.

<table>
<thead>
<tr>
<th>n</th>
<th>compares</th>
<th>moves</th>
<th>$n \log n$</th>
<th>$n^2/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>355</td>
<td>855</td>
<td>664</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>932</td>
<td>1932</td>
<td>1528</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>2266</td>
<td>4666</td>
<td>3457</td>
<td>80000</td>
</tr>
<tr>
<td>800</td>
<td>5216</td>
<td>10816</td>
<td>7715</td>
<td>320000</td>
</tr>
<tr>
<td>1600</td>
<td>11942</td>
<td>24742</td>
<td>17030</td>
<td>1280000</td>
</tr>
</tbody>
</table>

### 31 Heaps: a kind of tree

- Heap property: the value at a node is $\leq$ the value of each child.
- The smallest value is therefore at the root.

**Lecture 9, 2/13/2020**
• All leaves are at the same level ±1.
• To insert
  • Place new value at “end” of tree.
  • Let the new value sift up to its proper level.
• To delete: always delete the least (root) element
  • Save value at root to return it later.
  • Move the last value to the root.
  • Let the new value sift down to its proper level.
• Storage
  • Store the tree in an array [1 . . . ]
  • leftChild[index] = 2*index
  • rightChild[index] = 2*index+1
  • the last occupied place in the array is at heapSize.
void siftUp (int heap[], int subjectIndex) {
    // the element in subjectIndex needs to be sifted up.
    heap[0] = heap[subjectIndex]; // pseudo-data
    while (1) { // compare with parentValue.
        int parentIndex = subjectIndex / 2;
        if (heap[parentIndex] <= heap[subjectIndex]) return;
        swap(heap, subjectIndex, parentIndex);
        subjectIndex = parentIndex;
    }
} // siftUp

int betterChild (int heap[], int subjectIndex, int heapSize) {
    int answerIndex = subjectIndex * 2; // assume better child
    if (answerIndex+1 <= heapSize &&
        heap[answerIndex+1] < heap[answerIndex]) {
        answerIndex += 1;
    }
    return(answerIndex);
} // betterChild

void siftDown (int heap[], int subjectIndex, int heapSize) {
    // the element in subjectIndex needs to be sifted down.
    while (2*subjectIndex <= heapSize) {
        int childIndex = betterChild(heap, subjectIndex, heapSize);
        if (heap[childIndex] >= heap[subjectIndex]) return;
        swap(heap, subjectIndex, childIndex);
        subjectIndex = childIndex;
    }
} // siftUp
// intermediate algorithms

```c
void insertInHeap (int heap[], int *heapSize, int value) {
    *heapSize += 1; // should check for overflow
    heap[*heapSize] = value;
    siftUp(heap, *heapSize);
} // insertInHeap

int deleteFromHeap (int heap[], int *heapSize) {
    int answer = heap[1];
    heap[1] = heap[*heapSize];
    *heapSize -= 1;
    siftDown(heap, 1, *heapSize);
    return (answer);
} // deleteFromHeap
```

// advanced algorithm

```c
void heapSort(int array[], int arraySize) {
    // sorts array[1..arraySize] by first making it a heap,
    // then by successive deletion. Deleted elements go
    // to the end, leading to reverse sorting.
    int index, size;
    array[0] = -infinity; // pseudo-data
    // The second half of array[] satisfies the heap property.
    for (index = (arraySize+1)/2; index > 0; index -= 1) {
        siftDown(array, index, arraySize);
    }
    for (index = arraySize; index > 0; index -= 1) {
        array[index] = deleteFromHeap(array, &arraySize);
    }
} // heapSort
```

This method of heapifying is $\mathcal{O}(n)$:

- 1/2 the elements require no motion.
- 1/4 the elements may sift down 1 level.
- 1/8 the elements may sift down 2 levels.
• Total motion = \( (n/2) \cdot \sum_{1 \leq j} j/2^j \)
• That formula approaches \( n \) as \( j \to \infty \)

Experimental results for Heap Sort: compares + moves \( \approx 3.1n \log n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>compares</th>
<th>moves</th>
<th>( n \log n )</th>
<th>( n^2/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>755</td>
<td>1190</td>
<td>664</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>1799</td>
<td>2756</td>
<td>1528</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>4180</td>
<td>6196</td>
<td>3457</td>
<td>80000</td>
</tr>
<tr>
<td>800</td>
<td>9621</td>
<td>14050</td>
<td>7715</td>
<td>320000</td>
</tr>
<tr>
<td>1600</td>
<td>21569</td>
<td>31214</td>
<td>17030</td>
<td>1280000</td>
</tr>
</tbody>
</table>

32 Bin sort

• Assumptions: values lie in a small range, there are no duplicates.
• Storage: build an array of bins, one for each possible value. Each is 1 bit long.
• Space: \( O(r) \), where \( r \) is the size of the range.
• Place each value from input as a 1 in its bin. Time: \( O(n) \).
• Read off bins in order, reporting index if it is 1. Time: \( O(r) \).
• Total time: \( O(n + r) \).
• Total memory: \( O(r) \), which can be expensive.
• Can handle duplicates by storing a count in each bin, at a further expense of memory.

33 Radix sort

• Example: use base 10, with values integers 0 – 9999, with 10 bins, each holding a list of values, initially empty.
• Pass 1: insert values in bins (at end of list) based on their last digit.
• Pass 2: examine values in bin order, and in insertion order within bins, placing them in a new copy of bins based on second-to-last digit.
• Pass 3, 4: similar.
• The number of digits is \( O(\log n) \), so there are \( O(\log n) \) passes, each of which takes \( O(n) \) time, so the algorithm is \( O(n \log n) \).
34  Merge sort

```c
void mergeSort(int array[], int lowIndex, int highIndex) {
    if ((highIndex - lowIndex <= 1)) return;
    int mid = (lowIndex+highIndex)/2;
    mergeSort(array, lowIndex, mid);
    mergeSort(array, mid+1, highIndex);
    merge(array, lowIndex, highIndex);
} // mergeSort

void merge(int array[], int lowIndex, int highIndex) {
    int mid = (lowIndex+highIndex)/2;
    // copy the relevant parts of array to two temporaries
    // walk through the temporaries in tandem,
    // placing smaller in array
} // merge
```

- \( c_n = n + 2c_{n/2} \)
- \( a = 2, b = 2, k = 1 \Rightarrow O(n \log n) \).
  - This complexity is guaranteed.
- The sort is also stable: it preserves the order of identical keys.
- Insertion and selection sort are stable, but not quicksort or heapsort.

Experimental results for Merge Sort: compares + moves \( \approx 2.9n \log n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n \log n )</th>
<th>( n^2/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>546</td>
<td>1344</td>
</tr>
<tr>
<td>200</td>
<td>1286</td>
<td>3088</td>
</tr>
<tr>
<td>400</td>
<td>2959</td>
<td>6976</td>
</tr>
<tr>
<td>800</td>
<td>6741</td>
<td>15552</td>
</tr>
<tr>
<td>1600</td>
<td>15017</td>
<td>34304</td>
</tr>
</tbody>
</table>

35  Red-black trees (Guibas and Sedgewick 1978)

- Self-balancing trees during online insertion.
- Representation requires pointers both to children and to the parent.
• Each node is red or black.
• The pseudo-nodes (or null nodes) at bottom are black.
• The root node is black.
• Red nodes have only black children. So no path has two red nodes in a row.
• All paths from the root to a leaf have the same number of black nodes.
• For a node $x$, define black-height($x$) = number of black nodes on a path down from $x$, not counting $x$.
• The algorithm manages to keep height of the tree $\leq 2 \log(n + 1)$.
• To keep the tree acceptable, we sometimes rotate, which reorganizes the tree locally without changing the symmetric traversal.

![Diagram of rotations](image)

• To insert
  • place new node $n$ in the tree and color it red. $O(\log n)$.
  • walk up the tree from $n$, rotating as needed to restore color rules. $O(\log n)$.
case 1: parent and uncle red

Circled: black; otherwise: red
Star: continue up the tree here

case 2: parent red, uncle black, c inside

• try with values 1..6:
1 2 3
final color

1 2 3
case 3 color

1 2 3
rotate
case 1 color

1 2 3 4
rotate

case 1

1 2 3
rotate

case 3

• try with these values: 5, 2, 7, 4 (case 1), 3 (case 2), 1 (case 1)

36 Review of binary trees

• Binary trees have expected $O(\log n)$ depth, but they can have $O(n)$ depth.
• insertion
• traversal: preorder, postorder, inorder=symmetric order.
• deletion of node D
  • If D is a leaf, remove it.
  • If D has one child C, move C in place of D.
  • If D has two children, find its successor: $S = RL^*$. Move S in place of D. If S has no left child, but if it has a right child C, move C in place of S.

37 Ternary trees

• By example.
The depth of a balanced ternary tree is $\log_3 n$, which is only 63% the depth of a balanced binary tree.

The number of comparisons needed to traverse an internal node during a search is either 1 or 2; average 5/3.

So the number of comparisons to reach a leaf is $\frac{5}{3} \log_3 n$ instead of (for a binary tree) $\log_2 n$, a ratio of 1.05, indicating a 5% degradation.

The situation gets only worse for larger arity. For quaternary trees, the degradation is about 12.5%.

And, of course, an online construction is not balanced.

38 Quad trees (Finkel 1973)

- Extension of sorted binary trees to two dimensions.
- Internal nodes contain a discriminant, which is a two-dimensional $(x,y)$ value.
- Internal nodes have four children, corresponding to the four quadrants from the discriminant.
- Leaf nodes contain a bucket of $b$ values.

Insertion

- Dive down the tree, put new value in its bucket.
- If the bucket overflows, pick a good discriminant and subdivide.
- Good discriminant: one that separates the values as evenly as possible. Suggestion: median $(x, y)$ values.

Offline algorithm to build a balanced tree

- Put all elements in a single bucket, then recursively subdivide as above.

Generalization: for $d$-dimensional data, let each discriminant have $d$ values. There are $2^d$ children. Can become cumbersome when $d$ grows above about 3.

Heavily used in 3-d modeling for graphics, often with discriminant chosen as midpoint, not median.
39 k-d trees (Bentley and Finkel 1973)

- Extension of sorted binary trees to $d$ dimensions.
- Especially good when $d$ is high.
- Internal nodes contain a dimension number ($0 .. d − 1$) and a discriminant value (real).
- Internal nodes have two children, corresponding to values $\leq$ and $>$ the discriminant in the given dimension.
- Leaf nodes contain a bucket of $b$ values.
- Offline construction and online insertion are similar to quad trees.
  - To split a bucket of values, pick the dimension number with the largest range across those values.
  - Given the dimension, pick the median of the values in that dimension as the discriminant.
  - That choice of dimension number tends to make the domain of each bucket roughly cubical; that choice of discriminant balances the tree.
- Nearest-neighbor search: Given a $d$-dimensional probe value $p$, to find the nearest neighbor to $p$ that is in the tree.
  - Dive into the tree until you find $p$'s bucket.
  - Walking back up to the root, starting at the bucket:
    - If the domain of the other child of the node overlaps the ball, dive into that child.
    - If the ball is entirely contained within the node’s domain, done.
    - Otherwise walk one step up toward the root and continue.
  - Complexity: Initial dive is $O(n)$, but the expected number of buckets examined is $O(1)$.
- Used for cluster analysis, categorizing (as in optical character recognition).
40  2-3 trees

- By example.
- Like a ternary tree, but different rule of insertion
- Always completely balanced
- A node may hold 1, 2, or 3 (temporarily) values.
- A node may have 0 (only leaves), 2, 3, or 4 (temporarily) children.
- A node that has 3 values splits and promotes its middle value to its parent (recursively up the tree).
- If the root splits, it promotes a new root.

41  Stooge Sort

- A terrible method, but fun to analyze.

```c
#include <math.h>

void stoogeSort(int array[], int lowIndex, int highIndex){
    // highIndex is one past the end
    int size = highIndex - lowIndex;
    if (size <= 1) { // nothing to do
        return;
    } else if (size == 2) {
        if (array[lowIndex] > array[lowIndex+1]) {
            swap(array, lowIndex, lowIndex+1);
        }
    } else { // general case
        float third = ((float) size) / 3.0;
        stoogeSort(array, lowIndex, ceil(highIndex - third));
        stoogeSort(array, floor(lowIndex + third), highIndex);
        stoogeSort(array, lowIndex, ceil(highIndex - third));
    }
}
```

- Lecture 13, 2/27/2020
- $c_n = 1 + 3c_{2n/3}$
• \(a = 3, b = 3/2, k = 0\), so \(b^k = 0\). By the recursion theorem (page 18), since \(a > b^k\), we have complexity \(\Theta(n^{\log_b a}) = \Theta(n^{\log_{3/2} 3}) \approx \Theta(n^{2.71})\), so Stooge Sort is worse than quadratic.

• However, the recursion often encounters already-sorted sub-arrays. If we add a check for that situation, Stooge Sort becomes roughly quadratic.

## 42 Review

Insert the following items: 3 1 4 1 5 9 2 6 5 3 into:

• binary tree. Preorder result: 3 1 1 2 2 3 2 5 1 5 2 9 6
• top-light heap. Breadth-order result: 1 1 1 2 3 1 3 2 4 4 6 5 2 5 1
• array, then heapify. Breadth-order result: 1 1 1 2 3 1 3 2 4 6 5 2 5 1
• ternary tree. Preorder result: \((1, 3) 1 2 (2, 3) (4, 5) 5 2 (6, 9)\
• array, then 5 steps of selection sort. Result: 1 1 1 2 3 1 3 2 4 6 5 2 5 1
  Note: not stable.
• array, then 5 steps of insertion sort. Result: 1 1 1 2 3 1 4 5 1 9 2 6 5 2 3 2
• 2-3 tree. Preorder result: 3 1 1 (2, 3) 5 (4, 5) (6, 9)
• [Lecture 14, 3/3/2020]
• red-black tree. Preorder result: 3_b 1 1_b 2_b 3 5 4_b 5 9_b 6

## 43 B trees (Ed McCreight 1972)

• A generalization of 2-3 trees when McCreight was at Boeing, hence the name.

• Choose a number \(m\) (the \textbf{bucket size}) such that \(m\) values plus \(m\) disk indices fit in a single disk block. For instance, if a block is 4KB, a value takes 4B, and an index takes 4B, then \(m = 4\text{KB}/8\text{B} = 512\).

• \(m = 3 \Rightarrow 2\text{-3 tree}\).

• Each node has \(1 \ldots m - 1\) values and \(0 \ldots m\) children. (We have room for \(m\) values; the extra can be used for pseudo-data.)

• Shorthand: \(g = \lceil m/2 \rceil\).
• Internal nodes have \( g \ldots m \) children.

• Insertion
  - Insert in appropriate leaf.
  - If current node overflows (has \( m \) values) split it into two nodes of \( g \) values each; hoist middle value up one level.
  - When a node splits, its parent’s pointer to it becomes two pointers to the new nodes.
  - When a value is hoisted, iterate up the tree checking for overflow.

• B+ tree variant: link leaf nodes together for quicker inorder traversal. This link also allows us to avoid splitting a leaf if its neighbor is not at capacity.

• A densely filled tree with \( n \) keys (values), height \( h \):
  - Number of nodes \( a = 1 + m + m^2 + \cdots + m^h = \frac{m^{h+1} - 1}{m-1} \).
  - Number of keys \( n = (m - 1)a = m^{h+1} - 1 \Rightarrow \log_m(n + 1) = h + 1 \Rightarrow h \) is \( \mathcal{O}(\log n) \).

• A sparsely filled tree with \( n \) keys (values), height \( h \):
  - The root has two subtrees; the others have \( g = \lceil m/2 \rceil \) subtrees, so:
    - Number of nodes \( a = 1 + 2(1 + g + g^2 + \cdots + g^{h-1}) = 1 + \frac{2(g^h-1)}{g-1} \).
    - The root has 1 key, the others have \( g - 1 \) keys, so:
    - Number of keys \( n = 1 + 2(g^h - 1) = 2g^h - 1 \Rightarrow h = \log_g(n+1)/2 = \mathcal{O}(\log n) \).

44 Deletion from a B tree

• Insertion can overfl ow, causing a node to split.

• Deletion is only at leaves.

• Deletion can underflow, causing a node to have fewer than \( g \) keys.

• In case of underflow, borrow a value from a neighbor if possible, adjusting the appropriate key in the parent.

• If all neighbors (there are 1 or 2) are already minimal, grab a key from parent and also merge with a neighbor.
In general, deletion is quite difficult.

45 Hashing

- Very popular data structure for searching.
- Cost of insertion and of search is $O(\log n)$, but only because $n$ distinct values must be $\log n$ bits long, and we need to look at the entire key. If we consider looking at a key to be $O(1)$, then hashing is expected to be $O(1)$.
- Idea: find the value associated with key $k$ at $A[h(k)]$, where
  - $h()$ maps keys to integers in $0..s-1$, where $s$ is the size of $A[]$.
  - $h()$ is “fast”. (It generally needs to look at all of $k$, though.)
- Example
  - $k =$ student in class.
  - $h(k) =$ $k$’s birthday (a value from $0 .. 365$).
- Difficulty: collisions
  - Birthday paradox: $\text{Prob(no collisions with } j \text{ people)} = \frac{365!}{(365-j)365^j}$
  - This probability goes below $1/2$ at $j = 23$.
  - At $j = 50$, the probability is 0.029.
- Moral: One cannot in general avoid collisions. One has to deal with them.

46 Hashing: Dealing with collisions: open addressing

- Overview
  - The following methods store all items in $A[]$ and use a probe sequence. If the desired position is occupied, use some other position to consider instead.
  - These methods suffer from clustering.
Deletion is hard, because removing an element can damage unrelated searches. Deletion by marking is the only reasonable approach.

Perfect hashing: if you know all $n$ values in advance, you can look for a non-colliding hash function $h$. Finding such a function is in general quite difficult, but compiler writers do sometimes use perfect hashing to detect keywords in the language (like `if` and `for`).

Additional hash functions. Use a family of hash functions, $h_1(), h_2(), \ldots$

- insertion: key probing with different functions until an empty slot is found.
- searching: probe with different functions until you find the key (success) or an empty slot (failure).
- You need a family of independent hash functions.
- The method is very expensive when $A[]$ is almost full.

Linear probing. Probe $p$ is at $h(k) + p \pmod{s}$, for $p = 0, 1, \ldots$

- Terrible behavior when $A[]$ is almost full, because chains coalesce. This problem is called “primary clustering”.

Quadratic probing. Probe $p$ is at $h(k) + p^2 \pmod{s}$, for $p = 0, 1, \ldots$

- When does this sequence hit all of $A[]$? Certainly it does if $s$ is prime.
- We still suffer “secondary clustering”: if two keys have the same hash value, then the sequence of probes is the same for both.

Add-the-hash rehash. Probe $p$ is at $(p + 1) \cdot h(k) \pmod{s}$.

- This method avoids clustering.
- Warning: $h(k)$ must never be 0.

- [Lecture 16, 3/10/2020] Review after midterm
- [Lecture 17, 3/12/2020]
- Double hashing. Use two has functions, $h_1()$ and $h_2()$. Probe $p$ is at $h_1(k) + p \cdot h_2(k)$.

- This method avoids clustering.
- Warning: $h_2(k)$ must never be 0.
47 Interlude: Logistics

- Online classes will be held via Zoom.
  - The class meeting ID is 739546678.
  - You can get a copy of Zoom at uky.zoom.us.
  - To connect via your browser: https://uky.zoom.us/j/739546678
  - To connect by telephone: 646-876-9923
  - The password is 030135.

- Office hours will also be held via Zoom.
  - The office-hours meeting ID is 581609732.
  - To connect via your browser: https://uky.zoom.us/j/581609732
  - There is no password.
  - This meeting has a waiting room, so you might have to wait a while if someone else is meeting with me.

48 Hashing: Dealing with collisions: external chaining

- Each element in A is a pointer, initially null, to a bucket, which is a linked list of nodes that hash to that element; each node contains $k$ and any other associated data.
- insert: place $k$ at the head of $A[h(k)]$.
- search: look through the list at $A[h(k)]$.
  - optimization: When you find, promote the node to the start of its list.
- average list length is $s/n$. So if we set $s \approx n$ we expect about 1 element per list, although some may be longer, some empty.
- Instead of lists, we can use something fancier (such as 2-3 trees), but it is generally better to use a larger $s$. 
49 Hashing: What is a good hash function?

- Want it to be
  - Uniform: Equally likely to give any value in $0..s-1$.
  - Fast.
  - Spreading: similar inputs $\rightarrow$ dissimilar outputs, to prevent clustering. (Only important for open addressing, as described below.)

- Several suggestions, assuming that $k$ is a multi-word data structure, such as a string.
  - Add (or multiply) all (or some of) (or some of) (or some of) the words of $k$, discarding overflow, then mod by $s$. It helps if $s = 2^j$, because mod is then masking with $2^j - 1$.
  - XOR the words of $k$, shifting left by 1 after each, followed by mod $s$.

- Wisdom: The hash function doesn’t make much difference. It is not necessary to look at all of $k$. Just make sure that $h(k)$ is not constant (except for testing collision resolution).

50 Hashing: How big should $A$ be?

- Some open-addressing methods prefer that $s$ be prime.
- Computing $h()$ is faster if $s = 2^j$ for some $j$.
- Open addressing gets very bad if $s < 2n$, depending on method. Linear probing is the worst; I would make sure $s \geq 3n$.
- External chaining works fine even when $s \approx n$, but it gets steadily worse.

51 Hashing: What should we do if we discover that $s$ is too small?

- We can rebuild with a bigger $s$, rehashing every element. But that operation causes a temporary “outage”, so it is not acceptable for online work.
• Extendible hashing
  • Start with one bucket. If it gets too full (list longer than 10, say), split it on the last bit of \( h(k) \) into two buckets.
  • Whenever a bucket based on the last \( j \) bits is too full, split it based on bit \( j + 1 \) from the end.
  • To find the bucket
    • compute \( v = h(k) \).
    • follow a tree that discriminates on the last bits of \( v \). This tree is called a trie.
    • it takes at most \( \log v \) steps to find the right bucket.
    • Searching within the bucket now is guaranteed to take constant time (ignoring the \( \log n \) cost of comparing keys)

52 Hash tables (associative arrays) in scripting languages

• Like an array, but the indices are strings.
• Resizing the array is automatic, although one might specify the expected size in advance to avoid resizing during early growth.
• Perl has a built-in datatype called a hash.

```perl
my %foo;
foo{"this"} = "that".
```

• Python has dictionaries.

```python
Foo = dict()
Foo['this'] = 'that';
```

• JavaScript arrays are all associative.

```javascript
const foo = [];
foo['this'] = 'that';
```

53 Cryptographic hashes: digests

- purpose: uniquely identify text of any length.
- these hashes are *not* used for searching.
- goals
  - fast computation
  - uninvertable: given $h(k)$, it should be infeasible to compute $k$.
  - it should be infeasible to find collisions $k_1$ and $k_2$ such that $h(k_1) = h(k_2)$.
- examples
  - SHA-1: 160 bits, but (2005) one can find collisions in $2^{69}$ hash operations (brute force would use $2^{80}$)
  - SHA-2: usual variant is SHA256; also SHA-512.
- uses
  - storing passwords (used as a trap-door function)
  - catching plagiarism
  - for authentication ($h(m + s)$ authenticates $m$ to someone who shares the secret $s$, for example)
  - tripwire: intrusion detection

54 Graphs

- Our standard graph:

- Nomenclature
- **vertices**: \(V\) is the name of the set, \(v\) is the size of the set. In our example, \(V = \{1, 2, 3, 4, 5, 6, 7\}\).
- **edges**: \(E\) is the name of the set, \(e\) is the size of the set. In our example, \(E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}\).
- **directed graph**: edges have direction (represented by arrows).
- **undirected graph**: edges have no direction.
- **multigraph**: more than one edge between two vertices. We generally do not deal with multigraphs, and the word **graph** generally disallows them.
- **weighted graph**: each edge has numeric label called its **weight**.

- Graphs represent situations
  - streets in a city. We might be interested in computing **paths**.
  - airline routes, where the weight is the price of a flight. We might be interested in minimal-cost cycles.
    - Hamiltonian cycle: no duplicated vertices (cities).
    - Eulerian cycle: no duplicated edges (flights).
  - Islands and bridges, as in the bridges of Königsburg, later called Kaliningrad (Euler 1707-1783). This is a multigraph, not strictly a graph.

![Graph diagram](image)

Can you find an Eulerian cycle?

- Family trees. These graphs are **bipartite**: Family nodes and person nodes. We might want to find the shortest path between two people.
- Cities and roadways, with weights indicating distance. We might want a minimal-cost spanning tree.
55  Data structures representing a graph

- Adjacency matrix
  - an array $n \times n$ of Boolean.
  - $A[i, j] = \text{true} \Rightarrow$ there is an edge from vertex $i$ to vertex $j$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The array is symmetric if the graph is undirected
  - in this case, we can store only one half of it, typically in a 1-dimensional array
  - $A[i(i - 1)/2 + j]$ holds information about edge $i, j$.
  - Instead of Boolean, we can use integer values to store edge weights.

- Adjacency list
  - an array $n$ of singly-linked lists.
  - $j$ is in linked list $A[i]$ if there is an edge from vertex $i$ to vertex $j$.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 $\rightarrow$ 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 $\rightarrow$ 3 $\rightarrow$ 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2 $\rightarrow$ 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 $\rightarrow$ 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2 $\rightarrow$ 3 $\rightarrow$ 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
56 Computing the degree of all vertices

- Adjacency matrix: $O(v^2)$.

```c
foreach row (0 .. v-1) {
    degree[row] = 0;
    foreach col in 0 .. v-1 {
        if (A[row, col]) degree[row] += 1;
    }
}
```

- Adjacency list: $O(v + e)$.

```c
foreach row (0 .. v-1) {
    degree[row] = 0;
    for (other = A[row]; other != null; other = other->next) {
        degree[row] += 1;
    }
}
```

57 Computing the connected component containing node $i$ in an undirected graph

- why: to segment an image.
- method: depth-first search (DFS).

```c
void DFS(vertex here) {
    // assume visited[*] == false at start
    visited[here] = true;
    foreach next (successors(here)) {
        if (! visited[next]) DFS(next);
    }
} // DFS
```

- DFS is faster with adjacency list: $O(e' + v')$, where $e', v'$ only count to the number of edges and vertices in the connected component.
- DFS is slower with adjacency matrix: $O(v + v')$.
- For our standard graph (page 46), assuming that the adjacency lists are all sorted by vertex number (or that we use the adjacency matrix),
starting at vertex 1, we invoke DFS on these vertices: 1, 2, 3, 7, 6.

- Lecture 19, 3/26/2020
- DFS can be coded iteratively with an explicit stack

```c
void DFS(vertex start) {
    // assume visited[*] == false at start
    initializeStack(workStack);
    pushStack(workStack, start)
    while (! emptyStack(workStack)) {
        place = popStack(workStack);
        if (visited[place]) continue;
        visited[place] = true;
        foreach next (successors(place)) {
            if (! visited[next])
                pushStack(workStack, next); // could record "place" as parent
        } // foreach successor "next"
    } // while workStack not empty
}
```

58 To see if a graph is connected

- See if DFS hits every node.

```c
bool isConnected() {
    foreach j (vertices)
        visited[j] = false;
    DFS(0); // or any vertex
    foreach j (vertices)
        if (! visited[j]) return(false);
    return true;
} // isConnected
```

59 Breadth-first search

- applications
  - find shortest path in a family tree connecting two people
  - find shortest route through city streets
• find fastest itinerary by plane between two cities

• method: place unfinished vertices in a queue. These are the ones we still need to visit, in order.

```c
void BFS(vertex start) {
    // assume visited[*] == false at start
    initializeQueue(workQueue);
    visited[start] = true;
    insertQueue(workQueue, start)
    while (! emptyQueue(workQueue)) {
        place = deleteQueue(workQueue); // from front
        foreach next (successors(place)) {
            if (! visited[next]) {
                visited[next] = true;
                insertQueue(workQueue, next); // to rear
                // could record "place" as parent
                } // not visited
            } // foreach successor "next"
        } // while queue not empty
    } // BFS
```

• For our standard graph (page 46), assuming that the adjacency lists are all sorted by vertex number (or that we use the adjacency matrix), starting at vertex 1, BFS visits these vertices: 1, 2, 6, 3, 7.

• using adjacency lists, BFS is $O(v' + e')$.

60 Shortest path between vertices $i$ and $j$

• Compute BFS($i$), but stop when you visit $j$.

  • Actually, you can stop when you place $j$ in the queue.
  • Construct the path by building a back chain when you insert a vertex in the queue. That is, you insert a pair: (here, next).

• If edges are weighted:

  • Use a heap (top-light) instead of a queue. That’s why heaps are sometimes called priority queues.
  • stop when you visit $j$, not when you place $j$ in the queue.
61 Topological sort

- Lecture 20, 3/31/2020
- Sample application: course prerequisites place some pairs of courses in order, leading to a directed, acyclic graph (DAG). We want to find a total order; there may be many acceptable answers.

- Weiss §9.2

Possible results:

1. 4 10 1 2 6 5 7 8 3 9
2. 10 4 7 1 2 5 8 6 3 9

- method: DFS looking for sinks, which are then placed at the start of the growing result.

```plaintext
list answerList; // global

void topologicalSort() { // computes answerList
    foreach j (vertices) visited[j] = false;
    answerList = emptyList;
    foreach j (vertices)
        if (!visited[j]) tsRecurse(j);
} // topologicalSort

void tsRecurse(vertex here) { // adds to answerList
    visited[here] = true;
    foreach next (successors(here))
        if (!visited[next]) tsRecurse(next);
    insertHead(here, answerList);
} // tsRecurse
```
62 Dijkstra’s algorithm: Finding all shortest paths from given vertex in a weighted graph

The weights must be positive. [Weiss §9.3.2]

- Rule: among all vertices that can extend a shortest path already found, choose the one that results in a shortest path. If there is a tie ending at the same vertex, choose either. If there is a tie going to different vertices, choose both.

- This is an example of a greedy algorithm: at each step, improve the solution in the way that looks best at the moment.

- Starting position: one path, length 0, from start vertex $j$ to $j$.

<table>
<thead>
<tr>
<th>path</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$\rightarrow$ 0</td>
</tr>
<tr>
<td>5 6</td>
<td>$\rightarrow$ 40</td>
</tr>
<tr>
<td>5 3</td>
<td>120</td>
</tr>
<tr>
<td>5 1</td>
<td>60</td>
</tr>
<tr>
<td>5 1</td>
<td>$\rightarrow$ 60</td>
</tr>
<tr>
<td>5 6 3</td>
<td>100 better way to add vertex 3</td>
</tr>
<tr>
<td>5 6 4</td>
<td>160</td>
</tr>
<tr>
<td>5 1 2</td>
<td>140</td>
</tr>
<tr>
<td>5 6 3</td>
<td>$\rightarrow$ 100</td>
</tr>
<tr>
<td>5 6 4</td>
<td>160</td>
</tr>
<tr>
<td>5 1 2</td>
<td>140</td>
</tr>
<tr>
<td>5 6 3 4</td>
<td>$\rightarrow$ 120 better way to add vertex 4</td>
</tr>
</tbody>
</table>
• Another example: Start at 1.

<table>
<thead>
<tr>
<th>path</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\rightarrow 0)</td>
</tr>
<tr>
<td>1,2</td>
<td>(\rightarrow 3)</td>
</tr>
<tr>
<td>1,4</td>
<td>(\rightarrow 3)</td>
</tr>
<tr>
<td>1,2,3</td>
<td>(\rightarrow 4)</td>
</tr>
<tr>
<td>1,2,5</td>
<td>7</td>
</tr>
<tr>
<td>1,4,3</td>
<td>5</td>
</tr>
<tr>
<td>1,4,5</td>
<td>6</td>
</tr>
<tr>
<td>1,2,5</td>
<td>7</td>
</tr>
<tr>
<td>1,4,5</td>
<td>6</td>
</tr>
<tr>
<td>1,2,3,5</td>
<td>(\rightarrow 5)</td>
</tr>
</tbody>
</table>

63 Spanning trees

• Lecture 21, 4/2/2020

• Weiss §9.5

• **Spanning tree**: Given a connected undirected graph, a cycle-free connected subgraph containing all the original vertices.

• **Minimum-weight panning tree**: Given a connected undirected weighted graph, a spanning tree with least total weight.

• Example: minimum-cost set of roads covering a set of cities.
64 Prim’s algorithm

1. Start with any vertex as the current tree.
2. do $v - 1$ times
3. connect the current tree to the closest external vertex

- This is a **greedy algorithm**: at each step, improve the solution in the way that looks best at the moment.
- Example: start with 5. We add: (5,6), (5,1), (1,3), (3, 4), (1, 2)
- Implementation
  - Keep a heap of all external vertices based on their distance to the current tree (and store to which tree vertex they connect at that distance).
  - Initially, all distances are $\infty$.
  - Repeatedly take the closest vertex $f$ and add its edge to the current tree.
  - For all external neighbors $b$ of $f$, perhaps $f$ is a better way to connect $b$ to the tree; if so, update $b$’s information in the heap.
- Complexity: $O(v \log v)$

65 Kruskal’s algorithm

1. Start with all vertices, no edges.
2. do $v - 1$ times
3. add the lowest-cost missing edge that does not form a cycle

- This is a **greedy algorithm**: at each step, improve the solution in the way that looks best at the moment.
- We can stop when we have added $v - 1$ edges; all the rest will certainly introduce cycles.
- Data representation: List of edges, sorted by weight
- Complexity: assuming that keeping track of the component of each vertex is $O(\log^* v)$, the complexity is $O(e \log e + v \log^* v)$, because we must sort the edges and then add $v - 1$ edges.
66 Cycle detection: Union-find

• general idea
  - As edges are added, keep track of which connected component every vertex belongs to.
  - Any new edge connecting vertices already in the same component would form a cycle; avoid adding such edges.

• operations
  - Each vertex starts as a separate component.
  - union(b,c): assign b and c to the same component (for instance, when an edge is introduced between them).
  - find(b): tell which component b is in (if b and c are in the same component, don’t add an edge connecting them).

• method for union(b,c)
  - Every vertex has at most one parent, initially nil.
  - Find the representative b’ of b by following parent links until the end.
  - Find the representative c’ of c.
  - If b’ = c’, they are already in the same component. Done.
  - Point either b’ to c’ or c’ to b’ by introducing a parent link between them.
  - We want trees to be as shallow as possible. So record the height of each tree in its root. Point the shallower one at the deeper one.
  - We can compress paths while searching for the representative. In this case, the height recorded in the root is just an estimate.

• We use this data structure in Kruskal’s algorithm to avoid cycles:

```c
typedef struct {
  int name;      // need not be int
  struct vertex_s *representative; // NULL => me
  int depth;     // only if I represent my group; 0 initially
} vertex_t;
```
67  Numerical algorithms

- Lecture 22, 4/7/2020
- We will not look at algorithms for approximation to problems using real numbers; that is the subject of CS321.
- We will study integer algorithms.

68  Euclidean algorithm: greatest common divisor (GCD)

- Examples: gcd(12,60)=12, gcd(15,66)=3, gcd(15,67)=1.

```c
int gcd(a, b) {
    while (b != 0) {
        (a,b) = (b, a % b);
    }
    return a;
} // gcd
```

- Example: a 12 60 12  b 60 12 0
- Example: a 15 66 15 6 3  b 66 15 6 3 0
- Example: a 15 67 15 7 1  b 67 15 7 1 0

69  Fast exponentiation

- Many cryptographic algorithms require raising large integers (thousands of digits) to very large powers (hundreds of digits), modulo a large number (about 2K bits).
- to get $a^{64}$ we only need six multiplications: $((((a^2)^2)^2)^2)^2$
- to get $a^5$ we need three multiplications: $a^4 \cdot a = (a^2)^2 \cdot a$.
- General rule to compute $a^e$: look at the binary representation of $e$, read it from left to right. The initial accumulator has value 1.
  - 0: square the accumulator
• 1: square the accumulator and multiply by $a$.

• Example: $a^{11}$. In binary, 11 is expressed as 1011. So we get $(((1^2)a^2)^2 \cdot a)^2 \cdot a$, a total of 4 squares and 3 multiplications, or 7 operations. The first square is always $1^2$ and the first multiplication is $1 \cdot a$; we can avoid those trivial operations.

• In cryptography, we often need to compute $a^e \pmod{p}$. Calculate this quantity by performing $\text{mod } p$ after each multiplication.

• As we read the binary representation of $e$ from left to right, we could start with the leading 0’s without any harm.

• Example: $243^{745} \pmod{452}$. $745_{10} = 1011101001_2$.

```plaintext
1 a = 243
2 m = 452
3 r = 1
4 r = r^2*a \% m
5 r = r^2 \% m
6 r = r^2*a \% m
7 r = r^2*a \% m
8 r = r^2*a \% m
9 r = r^2 \% m
10 r = r^2*a \% m
11 r = r^2 \% m
12 r = r^2 \% m
13 r = r^2*a \% m
14 r
```

70 Integer multiplication

• The BigNum representation: linked list of pieces, each with, say, 2 bytes of unsigned integer, with least-significant piece first. (It makes no difference whether we store those 2 bytes in little-endian or big-endian.)

• Ordinary multiplication of two $n$-digit numbers $x$ and $y$ costs $n^2$.

[ Lecture 23, 4/14/2020 ]

• Anatoly Karatsuba (1962) showed a divide-and-conquer method that is better.
- Split each number into two chunks, each with \( n/2 \) digits:
  - \( x = a \cdot 10^{n/2} + b \)
  - \( y = c \cdot 10^{n/2} + d \)

  The base 10 is arbitrary; the same idea works in any base, such as 2.

- Now we can calculate \( xy = ac10^n + (bc + ad)10^{n/2} + bd \). This calculation uses four multiplications, each costing \((n/2)^2\), so it still costs \(n^2\). All the additions and shifts (multiplying by powers of 10) cost just \(O(n)\), which we ignore.

- But we can introduce \( u = ac, v = bd, \text{ and } w = (a + b)(c + d) \) at a cost of \((3/4)n^2\).

- Now \( xy = u10^n + (w - u - v)10^{n/2} + v \), which costs no further multiplications.

- Example
  - \( x = 3962, y = 4481 \)
  - \( a = 39, b = 62, c = 44, d = 81 \)
  - \( u = ac = 1716, v = bd = 5022, w = (a + b)(c + d) = 12625 \)
  - \( w - u - v = 5887 \)
  - \( xy = 17753722 \).

- We can apply this construction recursively. \( c_n = n + 3c_{n/2} \). We can apply the recursion theorem (page 16).
  - \( a = 3 \)
  - \( b = 2 \)
  - \( k = 1 \)
  - \( a > b^k, \text{ so } c_n = \Theta(n \log_a b) = \Theta(n \log_2 3) \approx \Theta(n^{1.585}) \).

- For small \( n \), this improvement is small. But for \( n = 100 \), we reduce the cost from 10,000 to about 1480. Running \( bc -l \):

```bash
1 power=l(3)/l(2)
2 a=100
3 e(power*l(a))
```
bigInt bigMult(bigInt x, y; int n) {
    // n-chunk multiply of x and y
    bigInt a, b, c, d, u, v, w;
    if (n == 1) return (toInt(x) * toInt(y));
    a = extractPart(x, 0, n/2 - 1); // high part of x
    b = extractPart(x, n/2, n-1); // low part of x
    c = extractPart(y, 0, n/2 - 1); // high part of y
    d = extractPart(y, n/2, n-1); // low part of y
    u = bigMult(a, c, n/2); // recursive
    v = bigMult(b, d, n/2); // recursive
    w = bigMult(bigAdd(a,b), bigAdd(c,d), n/2); // recursive
    return (bigAdd(
        bigShift(u, n),
        bigAdd(
            bigShift(bigSubtract(w, bigAdd(u,v)), n/2),
            v
        ) // add
    ) // add
    );
}

71 Strings and pattern matching — Text search problem

- The problem: Find a match for pattern $p$ within a text $t$, where $|p| = m$ and $|t| = n$.
- Application: $t$ is a long string of bytes (a “message”), and $p$ is a short string of bytes (a “word”).
- We will look at several algorithms; there are others.
  - Brute force: $\mathcal{O}(mn)$. Typical: $1.1n$ (operations).
  - Rabin-Karp: $\mathcal{O}(n)$. Typical: $7n$.
  - Knuth-Morris-Pratt: $\mathcal{O}(n)$ Typical: $1.1n$.
  - Boyer-Moore: worst $\mathcal{O}(mn)$. Typical: $n/m$.
- Non-classical version: approximate match, regular expressions, more complicated patterns.
72 Text search — brute force algorithm

- Return the smallest index $j$ such that $t[j..j + m - 1] = p$, or $-1$ if there is no match.

```c
int bruteSearch(char *t, char *p) {
    // returns index in t where p is found, or -1
    int n = strlen(t);
    int m = strlen(p);
    int tIndex = 0;
    p[m] = 0xFF; // impossible character; pseudo-data
    while (tIndex+m <= n) { // there is still room to find p
        int pIndex = 0;
        while (t[tIndex+pIndex] == p[pIndex]) // enlarge match
            pIndex += 1;
        if (pIndex == m) return (tIndex); // hit pseudo-data
        tIndex += 1;
    } // there is still room to find p
    return (-1); // failure
}
```

- Example: $p = "001", t = "010001$.
- Worst case: $O((n - m)m) = O(nm)$
- If the patterns are fairly random, we observe $O(n - m) = O(n)$.

73 Text search — Rabin-Karp

- The idea is to do a preliminary hash-based check each time we increment $tIndex$ and move on if there is no chance that this position works.
- Problem: how can we avoid $m$ accesses to compute the hash of the next piece of $t$?
- We will start with fingerprinting, a weak version of the final method, just looking at parity, and assuming the strings are composed of 0 and 1 characters.
The parity of a string of 0 and 1 characters is 0 if the number of 1 characters is even; otherwise the parity is 1.

- Formula: \( \text{parity} = \sum_j p[j] \pmod{2} \).

- We can compute the parities of windows of \( m (= 6) \) bits in \( t \). For example,

| \( j \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| \( t \) | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

- \( \text{tParity} \) 1 1 0 1 0 1 0 1 0 1 0 0 1 1

- Say that \( p = 010111 \), which has \( p\text{Parity} = 0 \). We only need to consider matches starting at positions 2, 4, 6, 8, 10, and 11.

- We have saved half the work.

- We can calculate \( t\text{Parity} \) quickly as we move \( p \) by looking at only 2, not \( p \), characters of \( t \):
  
  - Initially, \( t\text{Parity}_0 = \sum_{0 \leq j < m} t[j] \pmod{2} \).
  
  - Then, \( t\text{Parity}_{j+1} = t\text{Parity}_j + t[j] + t[j + m] \pmod{2} \)

```c
int fingerprintSearch(bit *t, bit *p) {
    int n = strlen(t);
    int m = strlen(p);
    int tIndex = 0;
    int pParity = computeParity(p, m);
    int tParity = computeParity(t, m);
    while (tIndex + m <= n) { // there is still room to find p
        if (tParity == pParity) { // parity check ok
            int pIndex = 0;
            while (t[tIndex + pIndex] == p[pIndex]) { // enlarge match
                pIndex += 1;
                if (pIndex >= m) return (tIndex);
            } // enlarge match
        } // parity check ok
        tParity = (tParity + t[tIndex] + t[tIndex + m + 1]) % 2;
        tIndex += 1;
    } // there is still room to find p
    return (-1); // failure
} // fingerprintSearch
```

- The update rule for \( t\text{Parity} \) can mask instead of computing mod by applying \& 1.
• Instead of bits, we can deal with character arrays.
  • We generalize parity to the exclusive OR of characters, which are just 8-bit quantities.
  • The C operator for exclusive OR is ^.
  • The update rule for tParity is
    \[ t\text{Parity} = t\text{Parity} \oplus t[t\text{Index}] \oplus t[t\text{Index}+m+1]; \]
  • We now have reduced the work to 1/128 (for 7-bit ASCII), not 1/2, for the random case, because only that small fraction of starting positions are worth pursuing.
• The full algorithm extends fingerprinting.
  • Instead of reducing the work to 1/2 or 1/128, we want to reduce it to 1/q for some large q.
  • Use this hash function for m bytes \( t[j] \ldots t[j+m-1] \):
    \[
    \sum_{0 \leq i < m} 2^{m-1-i}t[j+i] \pmod{q}.
    \]
    Experience suggests that q be a prime > m.
  • We can still update tParity quickly as we move p by looking at only 2, not p, characters of t:
    \[ t\text{Parity}_{j+1} = (t[j+m] + 2(t\text{Parity}_j - 2^{m-1}t[j])) \pmod{q}. \]
  • We can use shifting to compute tParity without multiplication:\n    \[ t\text{Parity}_{j+1} = (t[j+m]+(t\text{Parity}_j -(t[j] \ll (m-1)) \ll 1)) \pmod{q}. \]
    We still need to compute mod q, however.
• Monte-Carlo substring search
  • Choose q, a prime q close to but not exceeding \( mn^2 \). For instance, if \( m = 10 \) and \( n = 1000 \), choose a prime q near \( 10^7 \), such as 9,999,991.
  • The probability \( 1/q \) that we will make a mistake is very low, so just omit the inner loop. We will sometimes have a false positive, with probability, it turns out, less than \( 2.53/n \).
  • I don’t think we save enough computation to warrant using Monte Carlo search. If false positives are very rare, it doesn’t hurt to employ even a very expensive algorithm to remove them. Checking anyway is called the “Las-Vegas version”.

74 Text search — Knuth–Morris–Pratt

- Consider \( t = \text{Tweedledee} \) and \( \text{Tweedledum}, p = \text{Tweedledum} \).
- After running the inner loop of brute-force search to the \( u \) in \( p \), we have learned much about \( t \), enough to realize that none of the letters up to that point in \( t \) (except the first) are \( T \). So the next place to start a match in \( t \) is not position 1, but position 8.
- Consider \( t = \text{pappappappar}, p = \text{pappar} \).
- After running the inner loop of brute-force search to the \( r \) in \( p \), we have learned much about \( t \), enough to realize that the first place in \( t \) that can match \( p \) starts not at position 1, but rather in position 3 (the third \( p \)). Moving \( p \) to that position lets us continue in the middle of \( p \), never retreating in \( t \) at all.
- How much to shift \( p \) depends on how much of it matches when we encounter a mismatch in the inner loop. This shift table describes the first example.

\[
\begin{array}{ccccccccccc}
\text{p} & \text{p} & \text{a} & \text{p} & \text{p} & \text{p} & \text{a} & \text{r} \\
\text{k} & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{shift} & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]
- If our match fails at \( p[8] \), use shift[7]=8 to reposition the pattern.
- Here is the shift table for the second example.

\[
\begin{array}{ccccccccccc}
\text{p} & \text{T} & \text{w} & \text{e} & \text{e} & \text{d} & \text{l} & \text{e} & \text{d} & \text{u} & \text{m} \\
\text{k} & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{shift} & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]
- Try matching that \( p \) against \( t = \text{pappappapparrassanuaragh} \).
int KMPSearch(char *t, char *p) {
    int tIndex = 0;
    int pIndex = 0;
    int n = strlen(t);
    int m = strlen(p);
    char shiftTable[m];
    computeShiftTable(p, shiftTable);
    while (tIndex+m <= n) { // there is still room to find p
        while (t[tIndex+pIndex] == p[pIndex]) { // enlarge match
            pIndex += 1;
            if (pIndex >= m) return(tIndex);
        } // enlarge match
        tIndex += shiftTable[pIndex-1];
        pIndex = max(0, pIndex-shiftTable[pIndex-1]);
    } // there is still room to find p
    return(-1); // failure
} // KMPSearch

• Unfortunately, computing the shift table, although $O(m)$, is not straight-forward, so we omit it.
• The overall cost is guaranteed $O(n + m)$, but $m < n$, so $O(n)$.

75  Text search — Boyer – Moore simple

• [Lecture 25, 4/23/2020]
• Robert S. Boyer, J. Strother Moore (1977)
• We start by modifying bruteSearch to search from the end of $p$ backwards.
int backwardSearch(char *t, char *p) {
    int n = strlen(t);
    int m = strlen(p);
    int tIndex = 0;
    while (tIndex+m <= n) { // there is still room to find p
        int pIndex = m-1;
        while (t[tIndex+pIndex] == p[pIndex]) { // enlarge match
            pIndex -= 1;
            if (pIndex < 0) return(tIndex);
        } // enlarge match
        tIndex += 1;
    } // there is still room to find p
    return(-1); // failure
} // backwardSearch

• Occurrence heuristic: At a mismatch, say at letter $\alpha$ in $t$, shift $p$ to align the rightmost occurrence of $\alpha$ in $p$ with that $\alpha$ in the text. But don’t move $p$ to the left. If $\alpha$ does not occur at all in $p$, move $p$ to one position after $\alpha$.

• Initialize location array for $p$:

int location[256];

// location[c] is the last position in p holding char $c$
for (int charVal = 0; charVal < 256; charVal += 1) {
    location[charVal] = -1;
}
for (int pIndex = 0; pIndex < m; pIndex += 1) {
    location[p[pIndex]] = pIndex;
}

• Let $\alpha$ be the failure character, which is found at a particular $p$Index and $t$Index.

• Slide $p$: $t$Index += max(1, $p$Index - location[$\alpha$])

• This formula works in all cases.
  • $\alpha$ not in $p$ and $p$Index = m-1 ⇒ a full shift: $t$Index += m
  • $\alpha$ not in $p$ and $p$Index = j ⇒ a partial shift, larger if we haven’t travelled far along $p$: $t$Index += $p$Index + 1
  • $\alpha$ is in $p$. We shift enough to align the rightmost $\alpha$ of $p$ with the one we failed on, or at least shift right by 1.
• Examples
  • \( p = \text{rum}, t = \text{conundrum} \). We shift \( p \) by 3, another 3, and find the match.
  • \( p = \text{drum}, t = \text{conundrum} \). We shift \( p \) by 1, by 4, and find the match.
  • \( p = \text{natu}, t = \text{conundrum} \). We shift \( p \) by 2, then fail.
  • \( p = \text{date}, t = \text{detective} \). We would shift \( p \) left, so we just shift right 1, then 4, then fail.

• Match heuristic: Use a shift table (organized for right-to-left search).

• Use both the occurrence and the match heuristics, and shift by the larger of the two suggestions.

• Horspool’s version: on a mismatch, look at \( \beta \), which is the element in \( t \) where we started matching, that is, \( \beta = t_{i+m+1} \). Shift so that \( \beta \) in \( t \) aligns with the rightmost occurrence of \( \beta \) in \( p \) (not counting \( p_{m-1} \)).
  • This method always shifts \( p \) to the right.
  • We need to precompute for each letter of the alphabet where its rightmost occurrence in \( p \) is, not counting \( p_{m-1} \). In particular:
  • shift[\( \beta \)] = if \( \beta \) in \( p_{0..m-2} \) then \( m - 1 - \max\{j | j < m - 1, p_j = \beta\} \) else \( m \).

76 Advanced pattern matching, as in Perl

• Based on regular expressions; can be compiled into finite-state automata.

  • exact: \text{conundrum}
  • don’t-care symbols: \text{con.ndr.}
  • alternation: \text{c[ou]nund(rum|ite)}
  • repetition:
    • \text{c(on)*und}
    • \text{c(on)+und}
    • \text{c(on)\{4,5\}und}
  • character classes: \text{c\wnundrum\d\W}
  • pseudo-characters: ^\text{conundrum}$
• Beyond regular expressions in Perl
  • Reference to “capture groups”: con(un|an)dr\1m
  • Zero-width assertions: (?=conundrum)