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

Use

Implementation

Specification

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.

```
<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>create empty list</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert new node at front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete first node, returning data</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>count length</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search by data</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>sort</td>
<td>$O(n \log n)$ to $O(n^2)$</td>
</tr>
</tbody>
</table>
```
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, deques: 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.
```c
#define MAXSTACKSIZE 10
#include <stdlib.h>

typedef struct {
    int contents[MAXSTACKSIZE];
    int count; // index of first free space in contents
} stack;

stack *makeEmptyStack() {
    stack *answer = (stack *) malloc(sizeof(stack));
    answer->count = 0;
    return answer;
} // makeEmptyStack

void pushStack(stack *theStack, int data) {
    if (theStack->count == MAXSTACKSIZE) {
        (void) error("stack overflow");
    } else {
        theStack->contents[theStack->count] = data;
        theStack->count += 1;
    }
} // pushStack

int popStack(stack *theStack) {
    if (theStack->count == 0) {
        return error("stack underflow");
    } else {
        theStack->count -= 1;
        return theStack->contents[theStack->count];
    }
} // popStack

• 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.
#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.
```c
#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 \( O(1) \) (place at front)
  - `search(i)` is \( O(n) \) (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 \( O(n) \) (average: \( n/2 \) steps)
  - `search(i)` is \( O(n) \) (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.)

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

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

```c
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.

```c
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 $\mathcal{O}(\log n)$.
- 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 $\mathcal{O}(\log \log n)$. 
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

<table>
<thead>
<tr>
<th>when ( a )</th>
<th>( c_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &lt; b^k )</td>
<td>( \Theta(n^k) )</td>
</tr>
<tr>
<td>( a = b^k )</td>
<td>( \Theta(n^k \log n) )</td>
</tr>
<tr>
<td>( a &gt; b^k )</td>
<td>( \Theta(n^{\log_b a}) )</td>
</tr>
</tbody>
</table>

- 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.
  - \( \mathcal{O}(f(n)) \): 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

- Lecture 5, 1/30/2020
- Example with elicited values
- Pseudo-data: in the universal “null” node.
- \( \text{insert}(i) \) and \( \text{search}(i) \) are both \( \mathcal{O}(\log n) \) if we are lucky or data are random.
```c
#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) { // dive down tree
        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
        symmetric(tree->left);
        visit(tree);
        symmetric(tree->right);
    }
} // symmetric()
```

- **Pre-order traversal**

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

- **Post-order traversal**

```c
void postorder(treeNode *tree) {
    if (tree == NULL) { // do nothing
        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 $O(\log n)$, but we can generally treat them as $O(1)$. 
• 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:

```plaintext
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:

```plaintext
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 \( 1 \leq t \leq j \), 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)
• \( f(n) = n = \Theta(n^1) \)
• \( k = 1, a = 1, b = 2 \text{ (or } b = 3/2) \)
• \( a < b^k \)
• 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.

```c
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
<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>2</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>c</td>
<td>d</td>
<td>c</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>1</td>
<td>d</td>
<td>c</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>9</td>
<td>c</td>
<td>7</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>d</td>
<td>3</td>
<td>d</td>
<td>7</td>
<td>c</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>d</td>
<td>4</td>
<td>d</td>
<td>9</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
24 Using partitioning to select jth 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=pIAi7KcqMNU
- https://www.youtube.com/watch?v=qtdfW3TbeYY
- https://www.youtube.com/watch?v=wdcoRf8edM
27 Insertion sort

- Lecture 8, 2/11/2020

- Comb method:

  \[
  \begin{array}{c}
  \text{sorted} \\
  \text{probe} \\
  \text{unsorted}
  \end{array}
  \]

- \( n \) 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>
```c
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:

```
<table>
<thead>
<tr>
<th>sorted, small</th>
<th>unsorted, large</th>
</tr>
</thead>
</table>

```

- $n$ iterations.
- Iteration $i$ may need to search through $n - i$ places.
- $\Rightarrow O(n^2)$.
- Experimental results for Selection Sort:
  compares + moves $\approx n$. 
\[
\begin{array}{cccc}
  n & \text{compares} & \text{moves} & n^2/2 \\
  100 & 4950 & 198 & 5000 \\
  200 & 19900 & 398 & 20000 \\
  400 & 79800 & 798 & 80000 \\
\end{array}
\]

```c
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);
    }
} // 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>
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 $s$.

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

- 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

- Lecture 9, 2/13/2020

- Heap property: the value at a node is $\leq$ the value of each child.
- The smallest value is therefore at the root.
• 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.
// basic algorithms (top of heap is smallest element)

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
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
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 $O(n)$:

- **Lecture 10, 2/18/2020**
- 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);
}
```

```c
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
}
```

- \( c_n = n + 2c_{n/2} \)
- \( a = 2, b = 2, k = 1 \Rightarrow O(n \log n). \)
- 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>compares</th>
<th>moves</th>
<th>n log n</th>
<th>( n^2/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>546</td>
<td>1344</td>
<td>664</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>1286</td>
<td>3088</td>
<td>1528</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>2959</td>
<td>6976</td>
<td>3457</td>
<td>80000</td>
</tr>
<tr>
<td>800</td>
<td>6741</td>
<td>15552</td>
<td>7715</td>
<td>320000</td>
</tr>
<tr>
<td>1600</td>
<td>15017</td>
<td>34304</td>
<td>17030</td>
<td>1280000</td>
</tr>
</tbody>
</table>

### 35 Red-black trees (Guibas and Sedgewick 1978)

- [Lecture 11, 2/20/2020](#)
- Self-balancing trees during online insertion.
- Representation requires pointers both to children and to the parent.
• Each node is \textbf{red} or \textbf{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 \text{black-height}(x) = \text{number of black nodes on a path down from } x, \text{ not counting } x.
• The algorithm manages to keep height of the tree \(\leq 2 \log(n+1)\).
• To keep the tree acceptable, we sometimes \textbf{rotate}, which reorganizes the tree locally without changing the symmetric traversal.

\begin{center}
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (b) at (1,0) {b};
  \node (x) at (0.5,1) {x};
  \node (c) at (1.5,1) {c};
  \node (y) at (2,2) {y};
  \node (x') at (0.5,3) {x};
  \node (a') at (1.5,3) {a};
  \node (y') at (2,4) {y};
  \node (c') at (3,1) {c};

  \draw (a) -- (x);
  \draw (b) -- (x);
  \draw (x) -- (x');
  \draw (x') -- (a');
  \draw (x') -- (y');
  \draw (c) -- (y);
  \draw (c) -- (c');

  \draw (x') -- ++(0.5,0) node [above] {right};
  \draw (x') -- ++(-0.5,0) node [below] {left};
\end{tikzpicture}
\end{center}

• To insert
  \begin{itemize}
    \item place new node \([\text{n}]\) in the tree and color it red. \(O(\log n)\).
    \item walk up the tree from \([\text{n}]\), rotating as needed to restore color rules. \(O(\log n)\).  
  \end{itemize}
case 1: parent and uncle red

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

case 2: parent red, uncle black, c inside

case 3: parent red, uncle black, c outside

• try with values 1..6:
• 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.
  - Find the closest value in the bucket to \( p \). Cost: \( b \) distance measures. Result: a **ball** around \( p \).
  - 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));
    }
} // stoogeSort
```

- 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 5 1 5 6 2 9 1

• top-light heap. Breadth-order result: 1 1 2 3 1 5 2 9 4 6 5 2 5 1

• array, then heapify. Breadth-order result: 1 1 2 3 1 5 2 9 4 6 5 2 5 1

• ternary tree. Preorder result: (1, 3) 1 2 (2, 3) 5 4 (5, 1) 6 9

• array, then 5 steps of selection sort. Result: 1 1 2 3 1 5 2 9 4 6 5 2 5 1

Note: not stable.

• array, then 5 steps of insertion sort. Result: 1 1 2 3 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)

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 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 $m = 4KB/8B = 512$.

• $m = 3 \Rightarrow 2$-3 tree.

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

• Shorthand: $g = \lceil m/2 \rceil$. 

\[ \text{Lecture 13, 3/3/2020} \]
• Internal nodes have \( g \cdot m \) 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 \( O(\log n) \).

• 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 \( n = 1 + 2(g^h - 1) = 2g^h - 1 \Rightarrow h = \log_g(n+1)/2 = O(\log n) \).

44 Deletion from a B tree

• Insertion can \textbf{overflow}, causing a node to split.
• Deletion is only at leaves.
• Deletion can \textbf{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 \textbf{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
  • $k = \text{student in class}$. 
  • $h(k) = k$’s birthday (a value from $0$ .. $365$).

• Difficulty: collisions
  • Birthday paradox: $\text{Prob(\text{no collisions with } j \text{ people})} = \frac{365!}{(365-j)365^j}$
  • 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.