CPSC 319 Notes

Brian Pho

April 19, 2017

Contents

Chapter 1

Asymptotic Complexity

- Measure of growth rate
- Asymptote is a straight line approach by the curve
- Constants and lower order terms are not important for large n
- Care about asymptotic growth
- Upper and lower bounds are functions that are simple and tight to the original function

1.1 Big O Notation

- Specifies upper-bound
- *f(n)* = original function
- *g(n)* = upper-bound function
- *f(n)* is *O(g(n))* if there is a real constant

  $$c > 0$$

  and an integer constant

  $$n_0 \geq 1$$

  such that:

  $$f(n) \leq cg(n) \forall n \geq n_0$$

1.2 Classes of Algorithms

- O(1) - constant
- O(log n) - logarithmic
- O(n) - linear
- O(n log (n)) - n log n

- O(n^2) - quadratic
- O(n^3) - cubic
- O(2^n) - exponential

1.3 Best, Worst, and Average Case Complexities

- Found by considering all possible arrangements of inputs of size n
- Worse-case complexity - max # of steps taken
- Best-case complexity - min # of steps taken
- Average-case complexity - average # of steps taken

1.3.1 Example

- Statements that don't depends on n are O(1), constant time
- Ignore difference in execution time for simple statements
- Use worse case for conditional statement
- Sum Rule: if the complexity of a sequence of statements is the sum of 2 or more terms, discard the lower order term.
- Product Rule: if a process is repeated for each n of another process, then big-O is the product of the big-Os of each process.
- Steps that divide by 2 on each iteration, then O(log(n)). Base of log is steps.

Chapter 2

Search

- Data - information, facts, events
- Record - data pertaining to a unique object
- Field - a constituent part of a record
  -has type and size
- Key - the data field used to select or order records
- Primary Key - the first field for selecting or sorting
- Secondary Key - the field used if 2 or more records have equal primary keys
- Satellite Data - data in a non-key fields **not** used when searching or sorting

2.1 Sequential Search

- Start at the beginning, compare item to query key until we find a match or reach the end of the list
- Works on sorted and unsorted lists
- Can be used on arrays and linked lists
- Good for small search space
- Complexity: O(n)
- Where the algorithm has to traverse the entire array

2.1.1 Implementation

```
int sequentialSearch(int[] array, int key)
{
    for (int i = 0; i < array.length; i++)
    {
```

```
        if (array[i] == key)
        {
            return i
        }
    return -1;
    }
}
```

2.2 Binary Search

- Only works if array is sorted
- Steps:
  1. Divide array in half.
  2. If matches key, return.
  3. If key < middle item, divide the left array in half. Apply steps above recursively.
  4. If key > middle item, do step 3 but for right half.
  5. Keep halving until a match is found or can't subdivide anymore.
- Complexity: O(log n)
- Where the algorithm has to divide the array n times

2.2.1 Implementation

```
int binarySearch(int[] array, int key)
{
    int lo = 0, mid, hi = arr.length - 1;
    while (lo <= hi)
    {
        mid = (lo + hi) / 2;
        if (key < arr[mid])
        {
            hi = mid - 1;
        }
        else if (key > arr[mid])
        {
            lo = mid + 1;
        }
        else
        {
            return mid;
        }
    return -1;
    }
```

```
}
```

2.2.2 Recursive Implementation

```
int binarySearch(int[] arr, int first, int last, int key)
{
    if (first <= last)
    {
        int mid = (first + last) / 2;
        if (key == arr[mid])
        {
            return mid;
        }
        else if (key < arr[mid])
        {
            return binarySearch(arr, first, mid - 1, key);
        }
        else
        {
            return binarySearch(arr, mid + 1, last, key);
        }
    return -1
    }
}
```

2.3 Interpolation Search

- Similar to binary search but "midpoint" is set to where the item is likely
- Assume that the data in the array is sorted and uniformly distributed (Rarely true)

2.3.1 Implementation

```
double P = (double) (key - arr[lo]) / (arr[hi] - arr[lo]);
mid = lo + Math.ceil((hi - lo) * p);
```

- Complexity: O(log(log(n)))
- Not that great compared to binary search

Chapter 3

Sort

- Internal Sort: data kept in memory
- External Sort: data kept in secondary memory
- In-Place Sort: sorting done within an array, doesn't use extra memory
- Stable Sort: Preserves the relative order of equal keys
  If sort by name first then age, keeps alphabetical order

3.1 Analyzing Sorts

- Done by considering the number of comparisons and data movements
- Use big-O notation
- Efficiency may depends on initial order
- We measure the # of comparisons and data movements for the best, worst, and average case.
- The # of comparisons independent from data movements (comparisons are expensive)
- If data items are large, prefer sorts that minimize data movements.
- Moving large structs is expensive, moving external data even more so
- Sometimes simple, inefficient sorts are ok for *small data sets*

3.2 Ideal Performance of Sorts

$$O(n \log n)$$

3.3 Bubble Sort

- Works by swapping items if they are out of order
- Smallest item bubbles up to the top of the array on the first pass
- The next smallest item bubbles up to its proper spot
- Repeated until sort (nested loops)
- big-O is O(n^2)

3.3.1 Implementation

```
void bubbleSort(int[] arr)
{
    for (int i = 0; i < arr.length - 1; i++)
    {
        for (int j = arr.length - 1; j > i; j--)
        {
            if (arr[j] < arr[j - 1])
            {
                int temp = arr[j - 1];
                arr[j -1] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

3.4 Selection Sort

- Works by selecting the smallest item above the current item in the array and then swapping them
- Repeat for each item in the array, up to the second-last item
- After each pass of the outer loop, the low part of the array is sorted and is no longer considered
- big-O is O(n^2)

3.4.1 Implementation

```
void selectionSort(int[] arr)
{
    for (int i = 0; i < arr.length - 1; i++)
    {
        int min = i;
```

```java
        for (int j = i + 1; j < arr.length; j++)
        {
            if (arr[j] < arr[min])
            {
                min = j;
            }
        }
        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
}
```

3.5 Insertion Sort

- Start with 2nd item and compare it to the first item
  - If less than, move 1st to the right and insert 2nd
  - If more than, keep
- Repeat with each successive item, inserting it into its proper position
- Must move all items greater than item one position to the right
- big-O is O(n^2)

3.5.1 Implementation

```java
void insertionSort(int[] arr)
{
    for (int i = 1, j; i < arr.length; i++)
    {
        int temp = arr[i];
        for (j = i; j > 0 && temp < arr[j - 1]; j--)
        {
            arr[j] = arr[j - 1];
        }
        arr[j] = temp;
    }
}
```

3.6 Merge Sort

- Divide the array in half (equal size)
- Sort each sub array

  - Done by applying the merge sort recursively
  - Merge the sub-arrays into a temporary array
  - Copy temporary array back into original array

- big-O is O(n log n)

3.6.1 Implementation

```
void mergeSort(int[] arr, int first, int last)
{
    if (first < last)
    {
        int mid = (first + last) / 2;
        mergeSort(arr, first, mid);
        mergeSort(arr, mid + 1, last);
        merge(arr, first, mid, mid + 1, last);
    }
}
```

3.7 Quick Sort

- Choose one array element to be the pivot
- Partition the array into 2 subarrays, such that
  - Subarray1 contains only
$$elements \leq pivot.$$
  - The pivot is in its final position in the array
  - Subarray2 contains only
$$elements \geq pivot$$
  - Apply this procedure recursively to each subarray and stop when the subarray is less than or equal to 1 in length
- Try to choose pivots that divide the array into (nearly) equal halves
- Some possible approaches
  - Pick first element (poor if array is nearly sorted)
  - Pick the middle element
  - Pick the median of the first, middle, and last element
- To partition array:
  - Scan the array inward from the edges using two pointers

- Stop the left pointer when it reaches an element greater than the pivot
- Stop the right pointer when it reaches an element less than the pivot
- Exchange the two elements
- Repeat until the pointers cross

- During partitioning, the pivot is moved out of the way by exchanging with the first element. Then is moved back to its final position with another exchange.
- To avoid index bound checks, the largest element is put into the last array position. Done before the actual sort.

Chapter 4

Arrays and Linked Lists

4.1 Lists

- Are classified as linear data structures
- May be one-dimensional or multi-dimensional
- Each element of a list might consist of:
  - A single data item, or
  - A record or object (compound data)
- Lists may be ordered (sorted) or unordered
- A list is an Abstract Data Type (ADT) that supports these operations:
  - add(newEntry) new item to the end of the list
  - insert(newEntry, position) an item into a list at the specified position
  - delete(position) the item at the specified position
  - clear() deletes all items from the list
  - getEntry(position) returns the item at the specified position
  - replaceEntry(position, newEntry) overwrite the item at the specified position with a new item
  - getLength() returns the number of items currently in the list
  - isEmpty() returns true if no items in the list, false otherwise
  - isFull() returns true if the list is full, false otherwise
  - display() prints out all items in the list
  - contains(itemKey) returns true if the list contains the item, false otherwise
  - search(itemKey) returns the item that matches the key (or null if no match)
- Lists may be implemented in many ways (E.g. Arrays, Linked Lists, RAM, Secondary Storage)

4.2 Arrays

- Are also called physically ordered lists
- Definition: Are linear, random access data structures, whose elements are accessed by a unique identifier called an index or subscript. Elements are stored continuously (continuous memory chunks) in RAM or secondary storage.
- Most modern programming languages directly support arrays
- Structure:
  - Formal View
  - Set of array elements:
$$\{e_1, e_2, ..., e_n\}$$
  - Mapping function:
  - Set of indices:
$$\{i_1, i_2, ..., i_n\}$$
  - Programming View
  - Elements:
$$[e_1, e_2, ..., e_n]$$
  - Indices:
$$[i_1, i_2, ..., i_n]$$
  - Indices always start at 0
- Arrays are fixed in length
- Imposes a maximum size for a list (May run out of room or wastes space)
- Inserting an item into an array may require shifting elements to make room for it

  insert(newEntry, position) is *O(n)* in the worst case
- Deleting an item may require shifting items to fill the gap

  delete(position) is *O(n)* in the worst case
- Accessing an item by position is *O(1)* (getting/replacing entries is very quick)
- Must use sequential search on unordered arrays
- Can use sequential or binary search on ordered arrays
- A *vector* is an array that grows in size when it overflows (like in C++)

- A new array of size 2N is allocated
- All elements from the old array are copied to the new array
- The original reference is changed to point to the new array

- In addition to ordinary arrays, Java provides the class
  - java.util.Vector
  - java.util.ArrayList

4.3 Linked Lists

- Are also called logically ordered lists
- Definition: A linear data structure that consists of zero or more nodes (elements), where each node contains data and a pointer to the next node
- A head pointer is used to point to the first node of the list (the head)
- In the last node (the tail), the next pointer is set to *null* to signify the end of the list
- A linked list can grow and shrink without limit
- A node is allocated dynamically at run time whenever a new item is added to the list
- A node is freed (garbage collected in Java) whenever an item is deleted from the list
- A data field for a node may be
  - A primitive data type
  - A compound data type
  - A reference to an object
- In Java, each node is an object of a class such as the following:

```
public Class Node
{
    private double data;
    private Node next;

    public Node(double d, Node n)
    {
        data = d;
        next = n;
    }

    public void setNext(Node n)
    {
        next = n;
    }
```

```
    public Node getNext()
    {
       return next;
    }
    //Other methods here
}
```

- A linked list is initially empty, so set the head pointer to *null*
- When you add an item to the list, you create a new node object, and link it in
- To insert into the beginning of a list:
  - Allocate a new node and use a temporary variable to point to it
  - Set the *data* field to the desired value
  - Set the *next* pointer to the value in the head pointer
  - Set the head pointer to the temporary
- To insert into the middle or end of the list:
  - Find the predecessor node (may require search)
  - Use a variable to point to it
  - Allocate the new node (use temp variable to point to it)
  - Set the data field to the desired value
  - Set the next pointer to the value in the predecessor's next field
  - Set the predecessor's next pointer to the temporary variable
- To delete from the beginning of a list:
  - Set the *head* pointer to point to the successor node
  - Free the deleted node (automatically done in Java)
- To delete from the middle or end of a list:
  - Find the predecessor node (May require search and use a variable to point to it)
  - Set the predecessor's *next* pointer to the delete node's *next* value
  - Free the deleted node
- Be sure never to delete from an empty list (Check for null head pointer)
- A *tail pointer* points to the last node in the list (makes inserting and deleting to and from the end of the list easier)
- To *traverse* the linked list, follow the pointers until *null* is reached (Use a temporary pointer)
- Insertion and deletion are *O(1)*, once the position is known

  - However, if finding the predecessor node requires a search, this is *O(n)* in the average and worst case
  - Insertion and deletion using the head (or tail) pointer is *O(1)*

- Getting an entry at a pointer other than the head (or tail, if tail pointer used) requires sequential access (Is $O(n)$ in the average and worst case)

4.4 Doubly Linked Lists

- Enhance singly linked lists by adding pointers to predecessor nodes
- Allows list traversal from tail to head

```
public class Node
{
    private double data;
    private Node prev, next;

    // Constructor
    public Node(double d, Node p, Node n)
    {
        data = d;
        prev = p;
        next = n;
    }

    // Accessor methods
}
```

- Insertion and deletion are trickier to implement (especially at the start or end of the list, or when the list is, or about to become empty)
- Java provides a generic implementation with the class *java.util.LinkedList*

4.5 Circular Lists

- Are linked lists where the last node points to the first node
- Only a tail pointer is needed, since the head is pointed to by tail.next
- To insert into an empty list:
  - Allocate a new node
  - Set the data field to be the desired value
  - Assign the node's address to the tail pointer
  - Set the node's *next* field to the same value (a self reference)

- To insert into the end of the list:
  - Allocate a new node
  - Use a temporary pointer to point to it
  - Set the *data* field to the desire value
  - Set the *next* field to tail.next
  - Set tail.next to the temporary variable
  - Set tail to the temporary variable

4.6 Sparse Tables

- A table is normally implemented using a two dimensional array
- A sparse table has mostly empty cells (thus much space is wasted)
- Space can be saved by using:
  - One dimensional arrays of references for columns and rows
  - Linked list of nodes, where each node represents a filled cell. Each has data, a pointer to the next filled cell int he column, a pointer to the next filled cell in the row

Chapter 5

Stacks and Queues

5.1 Stack

- Are *last in, first out (LIFO)* queues
- Are linear data structures which can only be access at the "top" using *push and pop*
- Push: store an element on the top of the stack

  If the stack has a maximum size, you cannot push when the stack if full
- Pop: remove and return the element on the top of the stack

  You cannot pop an empty stack
- May implement additional operations to:
  - Return the top element without popping
  - Clear the entire stack
  - Check if the stack if full
  - Check if the stack is empty
- Stacks may be implemented using arrays
  - Must use a variable to point to the top
  - Will initially be set to -1 to indicate an empty stack.
  - E.g. int top $= -1$;
  - Will have a maximum size (unless a resizable array is used)
  - To push, increment top and store the element at that position in the array
  - E.g. array[++top] = elementValue;
  - To pop, copy the top element in the array to a temporary variable then decrement top
  - Return the value in the temporary variable
  - E.g. temp = array[top−−]; return temp;
- Stacks may also be implemented using linked lists

- Unlike arrays, have no maximum size
- To push, insert the element at the head of the list
- To pop, copy the element at the head of the list, delete the node, then return the element

- Push and pop are constant time operations (O(1))
- Java provides a generic implementation with the class java.util.Stack (extends vector so not a true stack)
- Allows access to elements now at the top

5.2 Queues

- Are analogous to lineups at store checkouts
- Are linear data structures that are *first in, first out (FIFO)* queues
- Elements can only be accessed at the head and tail of the list
- Have two basic operations:
  - *Enqueue*: Add an element to the end of the list (if the queue has a maximum size you cannot enqueue to a full queue)
  - *Dequeue*: Delete and return the element at the beginning of the list (you cannot dequeue from an empty queue)
- May implement additional operations to:
  - Return the first element without dequeuing
  - Clear the entire queue
  - Check if the queue if full
  - check if the queue is empty
- May be implemented using an array
  - Use two variables to point to the beginning and end of the list
  - The "head" index is incremented after dequeuing, the "tail" index when enqueuing
  - Since the indices will eventually run off the end, the array is "wrapped around" to form a circular array (ring buffer)
  - Modulus arithmetic must be used when incrementing the indices (Keep them in the range of 0 to N-1 where N is the size of the array)
  - Head and tail are set to -1 to indicate an empty queue
  - To enqueue:
    * If the queue is empty
    * Set head and tail to 0

    - Else (increment tail) and then mod N
    - Set array[tail] to element value
  - To dequeue:
    - Store array[head] in a temporary variable
    - If only one element in the queue (head == tail)
    - Set head and tail to -1 (indicates empty queue)
    - Else (increment head) and then mod N
    - Return the value in the temporary variable

- May be implemented using singly linked list
  - Unlike arrays, have no maximum size
  - To enqueue, insert the element at the tail of the list
  - To dequeue, copy the element at the head of the list, delete the node, then return the element

- Enqueue and dequeue are constant time operations *O(1)*

5.3 Priority Queues

- Are linear data structures that store *prioritized elements*

- Each element has an associated *priority*
  - Usually a numeric value, where the smallest value means the highest priority
  - Stored as a key in the node for an element

- When dequeuing, one always removes the element with the highest priority (lowest key) from the list

- May be implemented using an unsorted linked list
  - New elements are always added to the tail (Do the standard enqueue operation, is big-O *O(1)*)
  - To dequeue the highest priority element, one must search the entire list for the lowest key (Is *O(n)* in the best and worst cases)

- May be implemented using a sorted linked list
  - New elements are inserted into the list in their proper position using the key (Is *O(n)* in the worst case)
  - To dequeue the highest priority element, simply remove the first element (Is *O(1)*)

- Other possible implementations
  - Use a separate linked list for each priority group
  - Or references to the beginning and ends of sublists within a larger list
  - Use a type of binary tree called a heap

Chapter 6

Trees

6.1 Introduction

- A tree is a hierarchical data structure
- Is a collection of vertices (nodes) and edges (arcs)

  A vertex contains data and pointer information

  An edge connects 2 vertices
- A tree is drawn to grow downwards

  The root node is at the top of the structure
- Each node, except the root node, has only one node drawn above it, called the *parent* node
- A node may have zero or more *children*, drawn below it
- Nodes with the same parent are called *twins* or *siblings*
- Nodes with no children are called *leaf* nodes or *terminal* or *external* nodes
- Any node is the root of a *subtree*

  Consists of it and the nodes below it
- A set of trees is called a *forest*
- A tree consists of levels where root node is not counted as a level
- The *height (depth)* of a tree is the distance from the root to the node(s) furthest away
- The path length is the sum of edges from each node to the root
- With *ordered* trees, the order of the children at every node is specified
- Are much more useful than *unordered* trees

6.2 Binary Trees

- Are trees where every node has 0, 1, or 2 children
- Each node contains:
  - Data
  - A left child pointer
  - A right child pointer
  - A parent pointer (optional)
- A root pointer is used to point to the root node
- In Java, each node is an object of a class such as the following:

```
public class Node
{
    private int data;
    private Node parent, left, right;

    public Node(int el, Node p, Node l, Node r)
    {
        data = el;
        parent = p;
        left = l;
        right = r;
    }
}
```

6.3 Binary Search Trees

- Also called *ordered binary trees*
- Are binary trees organized so that:
  - Every left child is less than (or equal to) the parent node
  - Every right child is greater than the parent node
  - All nodes in any left subtree will be less than (or equal to) the parent node
  - All nodes in any right subtree will be greater than the parent node
- Different binary search trees can represent the same data
- The shape of the tree depends on the order of insertion
- In the best case, the tree is balanced and the height is minimized
- Height is approximately log(n)

- In the worst case, the tree degenerates into a linked list

  Height is n - 1

- If the tree is well balanced, searches are efficient

  O(log(n))

- Related to the binary search of an sorted array

6.3.1 Insertion

- Requires a search of the existing tree, to find the parent node of the new node
- New nodes are always added as leaf nodes
- The new node is then attached to the parent
- If the tree is empty, then the new node becomes the root node
- Iterative implementation

```
public void insert(int el, Node root)
{
    Node current = root, parent = null;

    while (current != null)
    {
        parent = current;
        if (el > current.data)
        {
            current = current.right;
        }
        else
        {
            current = current.left;
        }
    }

    if (root == null)
    {
        root = new Node(el, parent, null, null);
    }
    else if (el > parent.data)
    {
        parent.right = new Node(el, parent, null, null);
    }
    else
    {
        parent.left = new Node(el, parent, null, null);
    }
}
```

6.3.2 Traversal

- To traverse a tree, all nodes are *visited* once in some prescribed order
- Two types:
  * Depth-first: recursively visit each node starting at the far left (or right)
  * Breadth-first: starting at the highest level, move down level by level, visiting nodes on each level from left to right
- Can also start at the bottom, or traverse from right to left

6.3.3 Depth-first Traversal

Depth-first, in-order traversal

- Visits nodes in ascending order
- Implementation

```
public void inorder(Node current)
{
    if (current != null)
    {
        inorder(current.left);
        System.out.println(current.toString());
        inorder(current.right);
    }
}
```

- Would be called from the client code as follows:

```
Node root = null;
// Build tree doing successive insertion
...
// Traverse tree
inorder(root);
```

Depth-first, pre-order traversal

Processes the root node first, then the left subtree, then the right subtree

- Implementation:

```
public void preorder(Node current)
{
    if (current != null)
    {
        // visit current node; for example
        System.out.println(current.toString());
        preorder(current.left);
```

```
        preorder(current.right);
    }
}
```

Depth-first, post-order traversal

- Processes the left subtree, then the right subtree, then the node
- Implementation:

```
public void postorder(Node current)
{
    if (current != null)
    {
        postorder(current.left);
        postorder(current.right);
        // visit current node; for example
        System.out.println(current.toString());
    }
}
```

- The depth-first traversals could be implemented non-recursively
- Requires iteration and an explicit stack
- Less elegant than the recursive implementation

6.3.4 Breadth-first Traversal

- Requires use of a queue
- Top-down, left-to-right implementation

```
public void breadthFirst()
{
    IntBSTNode p = root;
    Queue queue = new Queue;
    if (p != null)
    {
        queue.enqueue(p);
        while (!queue.isempty())
        {
            p = (IntBSTNode) queue.dequeue();
            p.visit();
            if (p.left != null)
            {
                queue.enqueue(p.left);
            }
            if (p.right != null)
            {
```

```
                queue.enqueue(p.right);
            }
        }
    }
}
```

6.3.5 Searching

- Can be done iteratively:

```
public Node search(Node current, int key)
{
    while (current != null)
    {
        if (key == current.data)
        {
            return current; // found
        }
        else if (key < current.data)
        {
            current = current.left;
        }
        else
        {
            current = current.right;
        }
    }
    return null; // not found
}
```

- Is very efficient when performed on a "well-balanced" tree
- Is *O(log(n))* when the height of the tree is minimized
- Is also *O(log(n))* when the tree is formed by inserting nodes in random input orders
- Is less efficient if the tree has degenerated to a linked list which is *O(n)*

6.3.6 Deleting nodes

3 cases

- Deleting a leaf node

  Set the parent node's child pointer to null

  Free the deleted node's memory
- Deleting a node with only one child

  Set the parent node's child pointer to the child of the deleted node ("splice out" the node)

Free the deleted node's memory

– Deleting a node with two children

Find the smallest node in the right subtree below the node to delete

"Splice out" that node, using the steps form one form the cases above

Substitute the spliced node for the deleted node, either by copying or by adjusting pointers

Free the deleted node's memory

Note: could use the largest node in the left subtree for first step above

6.4 AVL Trees

6.4.1 The Need for Balanced Trees

- Searches and insertions are most efficient when a binary tree is well balanced.
- Binary trees may become unbalanced after insertions and deletions
- In the worst case, the tree degenerates into a linked list
- There are several variants of ordered binary trees that remain well balanced after insertions and deletions
- AVL trees are one example

6.4.2 Balance of a Node

- Definition: is the height of the right subtree minus the height of the left subtree:
- balance(n) = rightHeight(n) - leftHeight(n)
- where n is some node in the tree
- A negative balance means the tree is left-heavy and a positive balance means the tree is right-heavy

6.4.3 AVL Trees

- Named after their inventors
- Definition: is an ordered binary tree where every node has a balance of -1, 0, or +1
- Note that the difference between the subtrees can never exceed 1

6.4.4 Node Structure

- Must add a *balance* field to the Node class used for binary search trees

```
public class Node
{
    private int data;
    private Node parent, left, right;
    private int balance;
}
```

6.4.5 Insertion into an AVL Tree

- General procedure:
  1. Insert the node into the tree, following the rules for a regular binary search tree
  2. If necessary, adjust the shape of the tree so that it conforms to the rules of an AVL tree (involves doing a single or double rotation)
  3. Update the balance fields for all nodes affected by the steps above
- *Pivot Node*
- Definition: is the ancestor node closest to the inserted node that is *not* in balance (i.e not 0)
- It is possible there may be no pivot when doing an insertion
- One adjusts the AVL tree and updates the balances according to the nature of the pivot and where the insertion is done
- There are 3 possible cases when doing an insertion:
  1. There is no pivot
  2. The pivot exists, and you add to the shorter subtree
  3. The pivot exists, and you add to the longer subtree

Case 1: There is no pivot

- Essentially, you are adding to a subtree with all 0 balances
- You change the balances for all ancestor nodes by $\pm$ 1
- The shape of the tree is *not* adjusted after the insertion
- Procedure
  - Insert the node into its proper place in the tree
  - Adjust the balances for all nodes from the inserted node up to the root node (i.e. all nodes on the *search path*)

- The inserted node is given a balance of 0
- For the other nodes:

  If inserted node < node value, decrement balance

  If inserted node > node value, increment balance

Case 2: A pivot exists, and a node is added to the shorter subtree

- Essentially, you are adding to a shorter subtree to bring it into a better balance
- The shape of the tree is *not* adjusted after the insertion
- But the balances must be updated
- You must be able to tell if you are adding to the shorter subtree (to distinguish from Case 3); you are if
  - Pivot == +1 and inserted node < pivot node, or
  - Pivot == −1 and inserted node > pivot node
- Procedure:
  - Insert the node into its proper place in the tree
  - Adjust the balances for all nodes from the inserted node up to and including the *pivot* node
  - The inserted node is given a balance of 0
  - For the other nodes:

    If inserted node < node value, decrement balance

    If inserted node > node value, increment balance
  - Note that balances do not change above the pivot node

Case 3: A pivot exists, and you add to the longer subtree

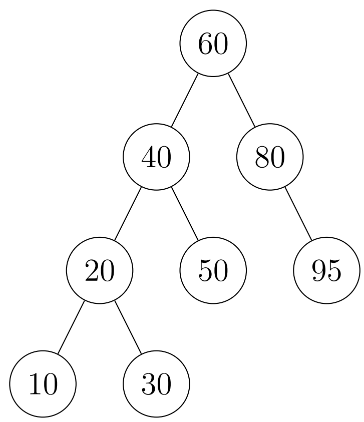
- Essentially, you are putting the tree into worse balance
- The pivot's balance changes to ± 2
- The shape of the tree *must* be adjusted after doing the insertion
- You must be able to tell if you are adding to the longer subtree (to distinguish from Case 2); you are if:
  - Pivot == +1 and inserted node > pivot node, or
  - Pivot == −1 and inserted node < pivot node
- Case 3 breaks down into 2 subcases:

  a) You add to the "outside subtree" of the "son" of the pivot on the search path

  b) You add to the "inside subtree" of the "son" of the pivot on the search path

Terminology

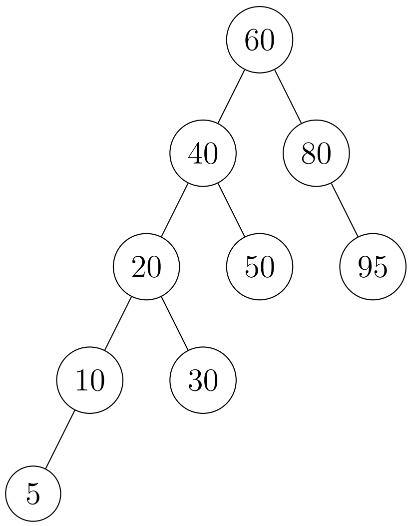
- *Ancestor node*: the parent node of the pivot node
- *Son node*: the child node of the pivot node, on the path from the pilot to the inserted node
- *Outside subtree*: The left subtree of the son, if the pivot is negative. The right subtree of the son, if the pivot is positive



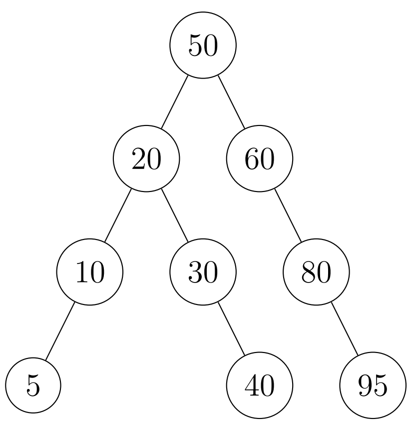
Case 3a: adding a node to the outside subtree

- Procedure:
  1. Insert the node into its proper place in the tree
  2. Adjust the shape of the tree by doing a *single rotation*. 2 cases:
     a) Do a *right rotation* if the outside subtree is on the left (the pivot is negative)
     b) Do a *left rotation* if the outside subtree is on the right (the pivot is positive)
  3. Adjust balances of affected nodes:
     a) Set pivot and insert node to 0
     b) Adjust the balances for all nodes above the inserted node, up to the child of the son node
     If inserted node $<$ node value, decrement balance
     If inserted node $>$ node value, increment balance
- 3 pointers must be changed:
  1. If pivot $<$ ancestor, then ancestor's *left* child pinter is set to the son node, otherwise set the *right* child pinter
  2. 2 cases:
     a) Right rotation: pivot's left child pointer set to the right child of the son node
     b) Left rotation: pivot's right child pointer set to the left child of the son node
  3. 2 cases:
     a) Right rotation: son's right child pointer set to the pivot node
     b) Left rotation: son's left child pointed set to the pivot node

Beginning of Tree



End of Tree **(FIX THIS)**



Case 3b: adding a node to the inside subtree

- *Grandson Node*: the child node of the son node, on the path from the pivot to the inserted node
- To adjust the tree after an insertion, a *double rotation* is performed; consists of:
  a) A right rotation at one node, followed by a left rotation at another node (RL rotation)
  b) The inverse (LR rotation)
- Procedure:
  1. Insert the node into its proper place in the tree
  2. Adjust the shape of the tree by doing a double rotation; 2 cases:
     a) RL rotation if the pivot is positive
     b) LR rotation if the pivot is negative
  3. Adjust the balances of affected nodes
     a) Set inserted node to 0

b) RL rotation

If inserted node > grandson, set pivot to -1

Else set pivot to 0, son to $+1$

c) LR rotation is symmetrical to the above

d) Adjust balances for all nodes above the inserted node up to the child of the son or pivot

If inserted node < node value, decrement balance

If inserted node > node value, increment balance

- RL Rotation:

  1. Right rotation through son

     a) Set pivot's right child pointer to grandson node

     b) Set son's left child pointer to grandson's right subtree (if it exists)

     c) Set grandson's right child pointer to son node

  2. Left rotation through pivot

     a) If there is no ancestor, set the root pointer to grandson; if pivot > ancestor, set the ancestor's right child pointer to the grandson, else set the left child pointer

     b) Set pivot's right child pointer to grandson's left subtree (if it exists)

     c) Set grandson's left child pointer to pivot

     Note: update parent pointer as you go

- LR rotation is symmetrical to the RL rotation

6.5 Midterm Info

- Format: Two parts
- MC part (have us trace code and complexity analysis of code)
- Three written answer question (complexity analysis, implementation of a simple data structure + methods, visual question = diagram of a data structure + adding/deleting)
- Everything up to and including tree (up to binary trees, not AVL trees)
- Know big omega and big theta
- Not expected to memorize and reproduce advance sorting algorithms
- Memorize big-O of sorting and searching algorithms
- Know stacks and queues extremely well

Chapter 7

Graphs

7.1 Classification

- Graphs are data structures where each node may have many predecessors and many successors.
- Are a generalization of tree structures, which generalize linear structures

7.2 Definition

- A graph consists of
  - A non-empty set of vertices (nodes) and,
  - A (possibly empty) set of edges (arcs) that connect vertices
- Set of vertices is denoted with V
  - V = $\{v_1, v_2, ..., v_n\}$
  - $|V|$ is the number of vertices in the set
- The set of edges is denoted with E
  - Each edge is a pair of vertices from V:
  - E.g. (v_0, v_2)
  - The set is a list of edges connecting vertices
  - E.g. $E = \{(v_1, v_3), (v_1, v_2), (v_0, v_2), (v_0, v_3)\}$
  - $|E|$ is the number of edges in the set
- A graph is denoted with G = (V, E)
- If the edge pair is unordered, then the graph is said to be undirected
  the path between vertices is bidirectional
- If the edge pair is ordered, the graph is a directed graph (digraph)
  - The path between vertices is unidirectional

- On diagrams, the edges are shown with arrows

- Two vertices are adjacent if an edge directly connects them
  - The vertices are called end vertices (or endpoints)
    If directed, the first endpoint is the origin, and the other is the destination
  - The neighbours of a vertex are all vertices that are adjacent to it
- An edge is incident on a vertex if the vertex is one of the edge's endpoints
  - The degree of a vertex v is the number of incident edges on v
    Denoted $\deg(v)$
  - If $\deg(v) = 0$, then v is an isolated vertex
    Since E can be empty, a graph can consist entirely of isolated vertices
- Two or more edges connecting the same 2 vertices are called parallel or multiple edges
  A graph containing these is a multigraph
- An edge connecting vertex to itself is called a *self-loop*
  A graph containing loops is called a *pseudograph*
- A *simple graph* contains no loops or parallel edges
- An edge may have a weight (or edge cost) that measures that cost of traversing it
  - Are positive integers or reals
  - A graph that incorporates weights is a *weighted graph*
  - On diagrams, edges are labeled with their weights
- A path is a sequence of vertices connected by edges
  E.g. v_0, v_2, v_3
- The (unweighted) path length is the number of edges on the path
  Is 2 for the above example
- The weighted path length is the sum of costs of edges on the path
  9 for the above example
- In a *simple path*, each vertex occurs only once in the sequence
- A *circuit* (cycle) is a path that begins and ends at the same vertex, and no edges are repeated
  However, vertices may be repeated
- A (simple) cycle is a circuit where all vertices (except the first and last) are different
- In a *connected graph*, there is a path between every pair of distinct vertices
- In a *complete graph*, there is one edge connected every pair of vertices
- A *directed acyclic graph* (DAG) is a diagraph containing no cycles

7.3 Operations on Graphs

Standard operations on the ADT

- Create: Set up on empty graph
- Clear: removes contents of the graph
- Insert node: add a vertex to the graph
- Insert edge: connect one vertex to another
- Delete node: remove a node (and incident edges) from the graph
- Delete edge: remove a connection between nodes

- Retrieve: get data stored in a node
- Update: overwrite data for a node
- Traverse: process each node in a specified order
- Find node: search for a particular node
- Find edge: search for an edge between nodes

7.4 Graph Representation

7.4.1 Adjacency Matrix

- Is $|V|X|V|$ 2D array
- Each row and column is labeled with a vertex
- 1 indicates an edge connecting vertices
- 0 indicates no edge

7.4.2 Adjacency List

- Uses a linked list for all the vertices
- Each vertex has its own linked list showing adjacent vertices

7.5 Graph Traversals

7.5.1 Depth-First Traversal

Basic Idea

- Visit a vertex v
- Recursively visit each unvisited vertex adjacent to v
- Implicitly uses a stack
- After backtracking, traverse any unvisited vertices using the same recursive process
- Vertices must include a field to indicate if it has been visited
- Pseudocode at page 379-380
- This algorithm creates a tree (or set of trees) that include all vertices of the graph
- Called a *spanning tree*
- Edges include in this tree are called *forward edges* (or *tree edges*)
  Shown with solid lines
- Edges not included are called *back edges*
  Shown with dashed lines

Complexity

- Is $O(|V| + |E|)$ for the adjacency list
- Is $O(|V|^2)$ for the adjacency matrix

7.5.2 Breadth-First Traversal

Basic Idea

- Process first vertex
- Then process *all* its unvisited neighbour vertices
- Then all unlisted neighbours of neighbours, etc.
- Needs a queue
- Pseudocode at page 382

7.5.3 Path Between Two Vertices

To find a path between 2 vertices, modify the traversals

- Start at the origin vertex
- Each time a vertex is visited, determine if it is the destination vertex
  If so, stop and print out the path
- If the traversal ends without finding the destination, then no path exists
- Note: this may not find the *shortest* path

7.5.4 Dijkstra's Algorithm

- Finds the shortest weighted paths from a single source vertex to all other vertices in a weighted, directed graph
  - All weights must be non-negative
  - Must be a simple graph
- For each vertex V, we must keep track of:
  - Whether the vertex still needs to be processed: toBeChecked(v)
  - The current distance from the source for the shortest path found so far: currDist(v)
  - The predecessor vertex for the shortest path found so far: pred(v)
- Can be implemented:
  - By adding fields to a vertex class
  - With parallel arrays (or a table)
- Can use a min heap to store the vertices to be checked, ordered by current distance
  Dequeue() will return vertex with smallest currDist()
- Is an example of a greedy algorithm:
  - Always takes the best immediate, or local solution while finding an answer.
  - In first step of while loop, always choose the vertex with the smallest distance.
- Complexity is $O(|V|^2)$
  - The while loop iterates $|V|$ times
    Is $O(|V|)$
  - The inner loop iterates deg(v) times
    Is $O(|V|)$
  - Can be improved to $O((|E|) + |V|)|g|V|)$ if a heap is used
  - Could be used to find the shortest paths between all pairs
    Apply the algorithm with every vertex as a source
    Is $O(|V|^3)$
  - If negative weights are used, then use Ford's algorithm (not on final)

7.5.5 Minimum Spanning Trees

- A tree is a special case of a graph:
  - Is connected
  - Has no cycles
  - One vertex is chosen to be the root node
- A spanning tree contains every vertex of a connected graph
  - May be created from the graph using depth first or breadth-first traversals
  - Any cycles will be removed
  - Usually several spanning trees are possible for a given graph:
    Depends on the type and order of traversal, and source vertex chosen
- A minimum spanning tree is a spanning tree with the minimum sum of edge weights
  - Specifies the cheapest way to interconnect vertices
  - Practical applications:
    - Connect sites in a telephone network
    - Finding ideal airline connections
- There are many algorithms to find the MST (minimum spanning tree), including:
  - Kruskal's
  - Jarnik-Prim's
  - etc
- Kruskal's Algorithm
  - Basic idea:
  - Order the edges by weight
  - Check each edge in turn
  - Add it to the tree if it doesn't create a cycle
  - Stop when all vertices are connected (i.e.. the number of tree edges = $|V|-1$)
- (MST) Complexity is $O(|E||g|E|)$, are equivalently $O(|E||g|V|)$
  - Depends on the efficiency of:
  - The sort
  - Cycle detection

7.5.6 Topological Order

- Is a linear ordering of vertices such that vertex a precedes vertex b whenever a directed edge exists from a to b
- Can only be applied to *directed acyclic graphs*
- Applications
  - Taking a sequence of courses given certain courses prerequisites
  - Scheduling tasks given precedence constraints
- Diagram looks like a finite state machine diagram
- Often, several topological orders are possible
- A *sink* or *minimal vertex* is a vertex with no outgoing edges

  Must be at least one in a DAG (Directed Acyclic Graph)
- Topological sort
- Basic Idea

```
for i = 1 to |V|
    find a sink vertex v;
    num(v) = i; // Number the vertices
    remove v and all its incident edges
```

- Can be implemented using a variant of the recursive depth-first traversal(p,g, 405-406)

```
topologicalSorting(digraph)
    for all vertices v
        num(v) = TSNum(v) = 0;
    i = j = 1;
    while there is a vertex v such that num(v) == 0
        TS(v);
    output vertices according to their TSNum?s;

TS(v)
    num(v) = i++;
    for all vertices u adjacent to v
        if num(u) == 0
            TS(u);
        else if TSNum(u) == 0
            error;
        TSNum(v) = j++;
```

Chapter 8

Hash Tables

8.1 Introduction

- Hash tables are classified as set structures
- Nodes have no predecessors or successors
- Are good for implementing dictionaries, where we only need the operations:
  - Insert
  - Delete
  - Search
- Are *not* suitable when we need to find an item's predecessor or successor
- Are not good for ordered lists
- With arrays, we directly access an array element using an index
- Hash tables are generalization of ordinary arrays:
  - Given a key K, we address the array to access element *array[K]*
  - If the key corresponds directly to the array index, then we have an ordinary array
  - Unlikely we'll be this lucky!
  - With hash tables, we *compute* this array index from the key using a *hash function*
  - i.e. Access an element using *array[h(K)]]*, where h is the hash function
  - Since accessing an element does not depend on n, the search operation takes constant time
  - i.e. is *O(1)*
  - Must better than:
    - Linear search: *O(n)*
    - Binary search: *O(log n)*

8.2 Hash Functions

- Transforms a key into a table address (array index):
- This table address is used to access an element in the hash table
- We say that key K_1 "hashes to " the address $h(K_1)$
- The table contains the key's satellite data

  And may duplicate the key itself
- Some slots in the table may not be used
- Ideally, the table size matches the number of elements to store
- A *collision* is where 2 or more keys hash to the same address
  - The keys are *synonyms*
- A *perfect hash function* transforms the set of keys into distinct addresses

  Does not result in collisions

  Is the ideal, but may be hard to find
- The choice of hash function depends on:
  - The nature of the keys
  - How densely one wishes to pack the table
  - The desire to avoid collisions
- In general, design the function so that any given key is equally likely to has to any of the m slots
  - Is randomly distributed
  - Known as *simple uniform hashing*
- Many hashing algorithms are possible; most have 3 steps:
  1. Represent the key in numerical form
     - If already a number, this step is done
     - If a string, take the ASCII code for each character to form a sequence of numbers
  2. Do arithmetic operations on the number to produce another number
     - E.g. Fold and add
     - $7679 + 8769 + ... + 3232 + 3232$ yields 33820
     - If the result produces arithmetic overflow, take the modulus after each addition

       Best to use a prime number for a more random distribution
     - Use mod 19937 for the above example, giving result of 13883
  3. Divide by the size of the address space m, and take the remainder

- Take the modulus
- The result will be in the range of 0 to m - 1
- Best to make m some prime number, to evenly distribute the resulting addresses
- E.g. 13883 mod 101 yields 46

- There are many methods possible for step 2 above
  - Folding and adding (shift folding) (shown above)
    - Can fold into other sizes before adding such as:
    - $767987 + 697676 + 323232 + 323232$
  - Reverse folding and adding (boundary folding)
    - Reverse the digits in some parts before adding
    - E.g. Key is 734211
    - Fold: 734 211 (arbitrary split between 4 and 2)
    - Add $734 + 112 = 846$
  - Mid-square method
    - Square the key and use the middle digits
    - E.g. Key is 068
    - $068^2 = 04624$ yields 62
  - Bit or digit extraction
    - Select bits or digits based on their randomness
    - E.g. Given the keys: 068, 160, 136, 092, 101, 127
    - Take the right two digits from every number in the list since they are fairly random
  - Radix transformation
    - Convert the key, using a base other than base 10
    - E.g. Key is 453 (base 10)
    - Interpret as base 11: $4 * 11^2 + 5 * 11^1 + 3 * 11^0 = 542$

8.3 Collision Resolution

- Several strategies are possible:
  - Find a hash function that avoids collisions altogether
    - Find a *perfect hash function*
    - Possible only for a static set of keys
    - Usually very difficult to find
  - Reduce the number of collisions to an acceptable number:
    - Find a function that hashes keys fairly randomly among the available addresses
      i.e. Avoid *clustering*

    - Increase the table size, so that a smaller proportion of it is actually used to store elements

      This wastes space

  - Store more than one element at a single address; several methods are possible, including:
    - *Separate chaining*
    - *Bucket addressing*
  - Store a colliding item at an address other than the hashed address; several methods are possible, including:
    - *Open addressing*
    - *Coalesced chaining*

- Separate chaining (*external chaining*)
  - Each position in the table is a reference to a corresponding linked list
  - Elements are stored as nodes in the list
  - The table is called a *scatter table*
    - E.g. $[A, B, C, D, E, F, G, H, I] \rightarrow [1, 7, 7, 6, 6, 2, 5, 0, 6]$
    - Solution: Have an array where each element is a linked list, each number is the index of the array and the elements are the data stored.
    - The linked list at index 6 holds 3 nodes: I, E, and D
    - The linked list at index 7 holds 2 nodes: C and B
  - If the linked lists are kept short, searches are fast
  - If ordered, then unsuccessful searches terminate more quickly
  - Requires addition space for maintaining references
  - For n keys, n + *TableSize* references
  - May be too expensive for large n

- Bucket addressing
  - Each table element is a block of memory big enough to hold several items
  - Is not very space efficient
  - Buckets can overflow
  - New items must be stored in another bucket, using a variation of open addressing

- Open addressing
  - Collisions are resolved by finding an "open" table entry at a position other than the original hashed address
  - If position $h(K)$ is occupied, then other positions are *probed* until:
    - An open position is found,
    - The same positions are tried repeatedly, or
    - The table is full

- New positions are tried in a probing sequence:
  - $norm(h(K) + p(1))$, $norm(h(K) + p(2))$, ..., $norm(h(K) + p(i))$, ...
  - p is some probing function
  - i is the probe number
  - *norm* is a normalization function, usually division modulo the table size
- *Linear probing* uses the function $p(i) = i$
  - Sequentially searches all positions after $h(K)$ until an open position is found
  - Tends to create large clusters in the table, which slows insertion and search operations
- *Quadratic probing* uses the function $p(i) = (-1)^{i-1}((i + 1)/2)^2$
  - Gives the sequence:
  - $h(K)$, $h(K) + 1$, $h(K) - 1$, $h(K) + 4$, $h(K) - 4$, ...
  - All modulo *TSize*
  - *TSize* should be an odd number
  - Although large primary clusters are avoided, *secondary clusters* can build up
- *Double hashing* can be used to avoid secondary clustering
  - Uses the probing function $i * h_p(K)$, where h_p is a second hashing function
  - Gives the sequence:
  - $h(K)$, $k(K) + h_p(K)$, $h(K) + 2 * h_p(K)$, ...
  - All modulo *TSize*
  - *TSize* should be a prime number, so that all table positions are included in the sequence
- Coalesced chaining combines probing with chaining
  - Each entry in the table contains a link field
  - If used, it points to another entry in the table
  - -1 indicates the end of the chain
  - If a key is not at its hashed address, then one follows the links until found
  - When inserting new items that collide, one can put the item:
    - In the next available position in the table (similar to linear probing)
    - Or in the last available position in the table
    - Or in a reserved part of the table called the *cellar*

8.4 Measuring Hashing Performance

- Can be done with the following, once the hash table has been filled:
  - $loadfactor = \frac{numberofrecords}{tablesize}$
  - $hashefficiency = \frac{loadfactor}{avg.\#readsperrecord}$
- Try for a load factor of at least 80 - 85% full
- A hash efficiency of 60% is adequate

Chapter 9

Heaps and Heapsort

9.1 Heaps

- A *max heap* is a binary tree where:
  - The value of each node is $\geq$ the values of its children
  - The tree is *complete*
  - The tree is perfectly balanced
  - The left nodes in the last level are all pushed to the left
  - Height is *O(log n)*
  - The largest element is always the root node
- A *min heap* is similar except the value of each node is $\leq$ the value of its children
  - The root contains the smallest element
- Heaps are not perfectly ordered
  - The above properties ensure only that order is maintained through linear lines of descent
  - Lateral lines may be out of order
- Heaps are normally implemented using arrays (or vectors)
  - Elements are stored sequentially in the array:
  - Level by level from top to bottom, and
  - From left to right at each level
  - The root node is always at position 0
  - The position of the left child of a node at i is $2i + 1$, where the position is < n
  - The position of the right child of a node at i is $2i + 2$, where the position is < n
  - The position of the parent node at i is $\frac{i-1}{2}$ where $1 \leq i < n$
    Note: assumes integer division
- Heaps are often used to implement priority queues

- More efficient than linear structures
- O(log n) vs O(n)
- Enqueue procedure:
  - Add the new element to the end of the heap
    i.e. At the end of the array
  - If necessary, restore the heap property by swapping the element with its parent
    Repeat until proper position found, or is at the root
- Dequeue procedure:
  - Remove the root element
    Always has the highest priority (at the root)
  - Replace it with the last leaf node
  - If necessary, restore the heap property by swapping the root with its larger child
    Repeat until proper position is found, or it becomes a leaf node

- Sometimes we need to reorganize the contents of an array into a heap
  - E.g. For the heapsort
  - Top-down method:
    1. Start with empty heap
    2. Sequentially enqueue new elements
    3. Is *O(n log n)* in the worst case
  - Bottom-up method (better method):
    1. Start with the last non-leaf node
       Is at position $\frac{n}{2} - 1$
       Set index to this position
    2. If necessary, restore the heap property by swapping with the largest child
       Repeat until proper place found, or becomes a leaf node
    3. Repeat step 2 after decrementing index
       Stop once the root has been processed

9.2 Heapsort

- Is an in-place sort of an array
- Procedure:
  - Recognize the array into a heap
    Bottom-up method is the quickest
  - for $(i = n; i > 0; i - -)$
    Swap root with element i
    Puts the largest element at the end of the array, so is no longer considered
    Restore heap property for elements 0 to $i - 1$

- Is $O(n \log n)$ in worst and average cases
- Is $O(n)$ in the best case for an array containing identical elements

Chapter 10

B-Trees

10.1 Introduction

- A B-Tree is a multiway balanced tree, where each node can have:
  - Many keys, up to a specified limit
  - Many children, up to a specified limit
- Invented in 1972 by Rudolf Bayer and Ed McCreight
- Since each node can have many children, a B-tree can have an enormous number of nodes with a small overall height

  Thus the search path to any node is short
- Makes searching and insertion efficient
- B-trees are good for storing data in secondary storage (usually hard disk drives)
- The short tree height minimizes the number of *disk accesses*
- Remember: disk accesses are far slower than memory accesses
- The size of a node is tailored to fit a *block* (page) of disk drive memory
- A *block* is a contiguous chunk of bytes that forms the basic unit of access
  - The size is system dependent
  - Typically is 512, 1024, 2048 bytes
- Since one is forced to read at least one block at a time, it is best to make the node $\leq$ block size
- Often, the B-tree itself is put into an *index file*
  - And the data to be stored is put into a separate random access *data file*
  - Thus, the B-tree only contains keys and pointers

    No satellite data
  - The data file contains the satellite data, and may duplicate the keys

- B-trees remain well balanced after doing insertions and deletions (described later)
- Leaf nodes are all at the same height

10.2 B-Tree Order

- B-trees are classified according to their *order*
- Using Knuth's definition, the order specifies the maximum number of children for a node

  E.g. In an order 5 B-Tree, a node can have up to 5 children
- Note: Bayer & McCreight define the order to be the minimum number of *keys* allowed for an internal nodes
- A B-tree of order m has:
  - Root node:
    - Has 0, or 2 to m children
    - 1 to m - 1 keys
  - Internal nodes:
    - k number of children, and
    - k - 1, where $ceiling(m/2) \leq k \leq m$
  - Leaf nodes:
    - 0 children
    - k - 1 keys, where $ceiling(m/2) \leq k \leq m$
- E.g. B-tree of order 5
  - Root node:
    - 0, or 2 to 5 children
    - 1 to 4 keys
  - Internal nodes:
    - 3 to 5 children
    - 2 to 4 keys
  - Leaf nodes:
    - 0 children
    - 2 to 4 keys
- E.g. B-tree of order 9
  - Root node:
    - 0, or 2 to 9 children
    - 1 to 8 keys

- Internal nodes:
  - 5 to 9 children
  - 4 to 8 keys
- Leaf nodes:
  - 0 children
  - 4 to 8 keys

- Note that all nodes (except possibly the root node) will always be at least half full

  E.g. At least 2 keys in an order 5 B-tree

10.3 Node Structure

- Each node consists of:
  - A *count*, indicating the number of keys actually used in the node
  - A variable number of *keys*
  - A variable number of *pointers*. There are 2 types
    - Pointers to other nodes in the B-Tree
    - Pointers to records in the random access data file
- The keys and pointers are grouped into *entries*
  - There 1 to m - 1 entries in a node
  - A single entry consists of:
    - P_i: a pointer to nodes with keys $< K_i$
    - K_i: a key
    - R_i: a pointer to a record in the data file, associated with the key K_i
- A *final pointer* P_F is also contained in the node
  - Points to a node with keys $>$ the key in the last entry
- E.g.
- |C $|P_1K_1, R_1|\ldots|P_F|$

10.4 Insertion

- New keys are always inserted into a left node
  - i.e. to a node at the bottom level
  - Requires a search that starts at the root, and traverses downwards through the tree
- If the nodes has room for the key, simply add it

- Keep the keys in ascending order
- Increment the count

- If the leaf node to add to is full, it must be "split"
- To split a node:
  - Create a new sibling node
  - Put the smallest n keys into the left node
    * $n = ceiling(\frac{m}{2}) - 1$
    * Set the count to n
  - Put the largest n keys into the right node
    Set the count to n
  - Put the middle key into the node one level above, creating a new node if necessary
    * Keep keys in ascending order
    * Increment count
    * Adjust node pointers
  - If the node one level up is full, split it using the same technique (applied recursively)
- Inserting into a full root node is a special case
  - In addition to a new sibling node, a new root node is created when splitting
- Unlike BSTs and AVL trees, B-trees grow "upwards"
  - Since we always insert into a left node, new interior and root nodes are created as needed

10.5 Deletion

- During deletion, remember that all nodes, except the root, must always be half full
  - i.e. must always contain at least n keys, where $n = ceiling(\frac{m}{2}) - 1$
  - If a node is left with too few keys, then
    * One borrows a key from a sibling if possible, or
    * One merges the node with a sibling node (and sometimes a parent node)
- Deleting from a left node breaks down into there cases:
  1. If the nodes has more than n keys, simply delete the key
     - Fill in any "hole" by shuffling keys leftwards
     - Decrement the count
  2. If the node has only n keys, borrow from a sibling if possible
     (a) Borrow from the left sibling if it has more than n keys, and has $\geq$ keys than the right sibling

- Delete the key
- Rotate the remaining keys clockwise through the parent
- Adjust the sibling's count

(b) Or borrow from the right sibling if it has more than n keys, and has more keys than the left sibling

- Delete the key
- Rotate the remaining keys counter-clockwise through the parent
- Adjust the sibling's count

3. If the node and its siblings have only n keys, then

   (a) Combine with the left sibling and a parent key, or

   (b) Combine with the right sibling and a parent key

   - Note that the parent is left with one less key
   - If it is a non-root node and has less than n keys, then it must be adjusted by:
     * Borrowing from a sibling, or
     * Combining nodes

- To delete a key from a non-leaf node:

  1. Swap the key with the smallest key in the right subtree
     - With its *successor*
     - Note: Could swap the key with the largest key in the left subtree (its *predecessor*)
  2. Delete the key (in its new position) using the procedures described above (to delete from a leaf node)