data structures

Class Information

Lecture: 10:00-10:50 AM (MWF)
Location: VAC 1B20
Professor: Gupta
Credits: 4
Recitation: Tuesday, 8:00 AM
Recitation Location: ECES 114

Chapter 1 Notes

→ algorithms: set of inputs that transform to give an output

→ precondition: conditions that must be true prior to an algorithms execution

→ postcondition: expected changes after algorithm executes

→ cost: memory usage and runtime for the algorithm
  ↳ faster computer will have lower cost

→ big-O notation: provides the upper bound on how quickly two functions grow as the input size $n \to$ infinity
  ↳ n = size of program's input (any data bits)

Chapter 2 Notes

→ **bit**: smallest unit of info stored in a computer

→ individual bits grouped together in groups of 8 to create bytes

→ digits 0-9 in decimal & hex are the same; digits 10-15 are A-F in hex

→ maximum value of one byte is determined by setting all bit positions to 1 & summing them up

→ 15 is the max value of F

→ to convert:

↳ split into groups of 4 bits (nibbles) & calculate hex value

→ more than one byte of info for numbers larger than 255

→ **int**: 4 bytes

→ **char**: 1 byte

→ **float**: 4 bytes

→ **long**: 8 bytes

→ **double**: 8 bytes

→ **pointers**: variables that store the memory address of other variables

↳ declared as: int *p = &x (p points to the address of x)

↳ * indicates that var is being declared as ptr.

↳ & indicates "address of"

Chapter 3 Notes

→ **arrays:** data structure used to store a collection of data, where each element is of same type

→ all elements are stored in contigous block of memory

↳ makes it easy and fast to access

→ limitations:

↳ fixed size (needs array doubling algorithms for it)

→ algorithm to delete array element

```
for x = index → numElements - 2
    A[x] = A[x+1]
numElements = numElements - 1
```

→ when array is declared, the name of array is pointer to first element in the array in memory

→ when memory is statically allocated for a variable, the memory is reserved until the variable goes out of scope

→ when memory is requested dynamically, an address is returned & the memory needs to be accessed w/ a pointer

→ **stack memory:**

↳ local vars stored on stack

↳ limited space available

→ **heap:** pool of free memory used for storing variables created dynamically during runtime

↳ stays allocated until it is deallocated

↳ must be accessed through a pointer

→ to dynamically allocate variable, use new

→ array doubling (complexity O(n))

```
doubleArray(A)
B.length = A.length * 2
for x = 0 to A
    B[x] = A[x]
return B
```

Chapter 4 Notes

→ **sorting algorithm**: algorithm that puts the elements in collection in specified order

→ **bubble sort**: individual elements "bubble" to their correct location through a series of individual swaps

```
bubbleSort(A)
    for i=0 → A.end-1
        for j=0 to A.end-i-1
            if A[j] > A[j+1]
                swap=A[j]
                A[j] = A[j+1]
                A[j+1] = swap
```

→ **insertion sort**: elements moved into correct location in the array one at a time

```
insertionSort(A)
    for i=1 to A.end
        index = A[i]
        j=1
        while(j>0 & A[j-1]>index)
            A[j] = A[j-1]
            j = j-1
        A[j]=index
```

Chapter 5 Notes

→ linked list: data structure that allows for individual elements to be added and removed as needed

→ memory is allocated for individual elements + pointers link those elements together

→ two types: singly linked & doubly linked

↳ singly linked: each element (node) contains data stored in the node and a pointer to the next node in the list

↳ doubly linked: each node in the list contains the node data, pointer to next node in list, and pointer to previous node in list (for head, previous is set to NULL

→ head: node at beginning of list

→ tail: last node

→ building doubly linked list:

↳ create new node dynamically

↳ set values for previous, key, and next

↳ creare another node dynamically & set the values for key, next and previous of n1

↳ connect the nodes

n0 → next = n1

n1 → previous = n0

→ traversing a linked list:

↳ linked list nodes are accessed through the pointers stored in the list

```
traverse()
    tmp = head
    while (temp != null)
        print temp → key
        temp = temp → next
```

→ search function

```
search(value)
    temp=head
    returnNode=NULL
    found=false
    while(!found & tmp!=NULL)
        if(tmp→key==value)
            return=true
            returnNode=temp
        else
            tmp=temp→next
```

→ insert node into singly linked list

```
insertNode(leftvalue,value)
    left=Search(leftvalue)
    node→key=value
    if left==NULL  //head node
        node→next=head
        head=node
    else if left→next=node
        tail=node
    else
        node→next=left→next
        left→next=node
```

→ inserting node into doubly linked list

```
insertNodeDouble(leftvalue,value)
    left=searcn(leftvalue)
    node→key=value
    if left==NULL  // head
        node→next=head
        head.previous=node
```

```
        head = node
    else if left -> next == NULL  // tail
        left -> next = node
        node -> previous = left
        tail = node
    else
        left -> next -> previous = node
        node -> previous = left
        node -> next = left -> next
        left -> next = node
```

→ deleting a node from a singly linked list

```
delete(value)
    if(head -> key == value)
        temp = head
        head = head -> next
        delete tmp
    else
        left = head
        tmp = head -> next
        found = false
        while(temp != NULL && !found)
            temp -> key == value
            left -> next = temp -> next
            if temp == tail
                tail = left
            delete temp
            found = true
            else
                left = tmp
                tmp = tmp -> next
```

Chapter 6 Notes

→ **Stack**: data structure that stores a collection of elements one ↳ LIFO

→ class Stack

```
private
    top
    data
    maxSize
public
    Init()
    isFull()
    isEmpty()
    push(value)
    pop()
```

```
push(value)          // push array
  if(!isFull())
    data[top] = value
    top = top+1
```

```
push(value)          // push linked list
  node→key = value
  if(!isEmpty())
     node→previous = top
  else
     node→previous = NULL
  top = node
```

```
pop()                //pop array
  if(!isEmpty())
     top = top-1
  else
     print("err")
  return data[top]
```

```
pop()      //linked list
  if (!isEmpty())
     x = top
  else
     print("err")
  top = top→previous
  return x
```

→ complexity: O(1) for push/pop

Chapter 7 Notes

→ queue: like a stack but FIFO

→ class Queue

```
  private:
    head, tail, data, queueSize, maxQueue, isEmpty() isFull()
  public
    init()
    enqueue(value)
    dequeue()
enqueue(value)      // array
  if(!isFull())
    data[tail] = value
    queueSize++
    if (tail == maxQueue-1)
      tail = 0
    else
      tail++
  else
    print("Queue full")
dequeue()           // array
  if(!isEmpty())
    value = data[head]
    queueSize--
  if(head == maxQueue-1)
    head = 0
  else
    head++
return value
```

```
enqueue(value)    // linked list
  node->key = value
  node->next = NULL
  if(tail != NULL)
     tail->next = node
     tail=node
  else
     tail=node
     head=tail
dequeue()     // linked list
  IF(head != NULL)
     node=head
     head = head->next
  else
     print("queue empty")
     tail=head
  return node
```

Chapter 8 Notes

→ **binary trees**: similar to linked lists in that the nodes in the tree can be created dynamically and then linked together to create a structure

→ pointers in binary tree:
- ↳ parent
- ↳ left child
- ↳ right child

→ all binary trees are defined in the terms of smaller sub-trees within it (self-similarity)

Chapter 9 Notes

→ **binary search tree:** special case of binary tree where data in the tree is ordered

↳ for any node in the tree, the nodes in the left sub-tree of that node all have a value less than the node

→ BinarySearchTree

```
private:
    root
    searchRecursive(node, value)
public:
    init()
    insert(value)
    search(value)
    traverseAndPrint()
    delete(value)
    deleteTree()
```

```
struct Node
    int key
    node *parent
    node* leftChild
    node* rightChild
```

→ search(searchKey)

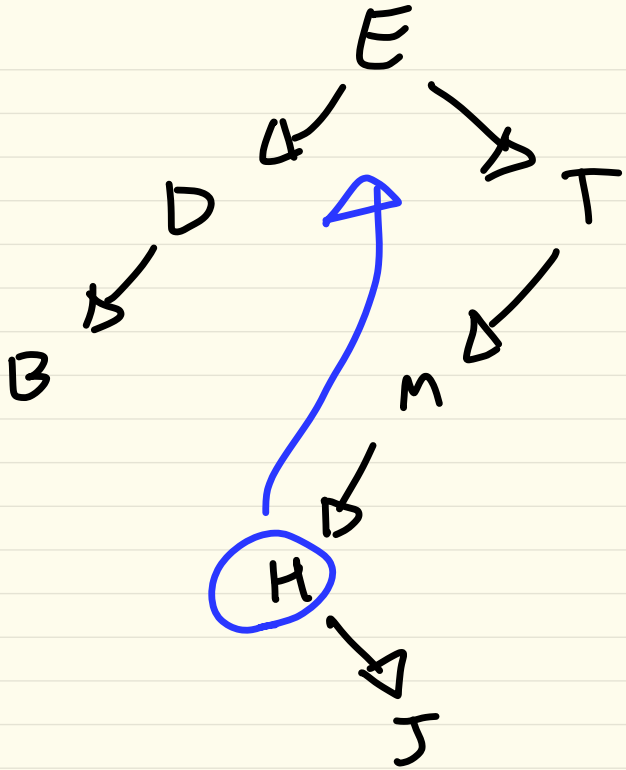
```
return searchRecursive(root, searchKey)
```

→ searchRecursive(node, searchKey)

```
if (node != NULL)
    if (node→key == searchKey)
        return node
    else if (node→key > searchKey)
        return searchRecursive(node→leftChild, searchKey)
    else
        return searchRecursive(node→rightChild, searchKey)
else return NULL
```

```
→ searchIterative (searchKey)
    node = root
    while (node != NULL)
        if (node→ key > searchKey)
            node = node→ leftChild
        else if (node→key < searchKey)
            node = node→ rightChild
        else
            return node
    return NULL
→ insert (value)
    tmp = root
    node→key = value
    node→parent = NULL
    node→ leftChild = NULL
    while (tmp != NULL)
        parent = tmp
        if(node→ key < tmp→ key)
            tmp = temp→ leftChild
        else
            tmp = tmp→ rightChild
    if (parent == NULL)
        root = node
    else if (node→key < parent→ key)
        parent→ leftChild = node
        node→ parent = parent
    else
        parent→ rightChild = node
```

```
→ delete(value)
    node = search(value)
    if(node != root)
        if(node→leftChild == NULL & node→rightChild == NULL)  // no children
            node→parent→leftChild = NULL
        else if(node→leftChild != NULL & node→rightChild = NULL)  // 2 children
            min = treeMinimum(rightChild)
            if(min == rightChild)
                node→parent→leftChild = min
                min→parent = node→parent
                min→leftChild = node→leftChild
            else
                min→parent→leftChild = min→RightChild
                                ... cont.
```


E
D
T
B
M
H
J


7 15 8 11 18 12 27
22 15 18 25 19 34
sum of

Chapter 8/9 Notes

Review of Trees

- binary trees: parent, left child, right child, root
- BST: data in tree is ordered

BST ADT

```
searchRecursive(node, searchKey)
  if(node != NULL)
    if(node->key == searchKey)
       return node;
    else if(node->key > search->key)
       return searchRecursive(node->left, searchKey)
    else
       return searchRecursive(node->right, searchKey)
  else
     return NULL;

searchIterative(searchKey)
  node = root
  while(node != NULL)
    if(node->key > searchKey)
       node = node->left;
    else if(node->key < searchKey)
       node = node->right;
    else
       return node
  return NULL;
```

```
insert(val)
  tmp = root;
  node->key = value;
  node->parent = NULL;
  node->leftChild = NULL;
  node->rightChild = NULL;
  while (tmp != NULL)
    parent = tmp
    if (node->key < tmp->key)
        tmp = tmp->leftChild
    else
        tmp = tmp->rightChild
  if (parent = NULL)          // if no parent, node becomes root
    root = node
  else if (node->key < parent->key)
    parent->left = node
  else
    parent->right = node
```

set vals to param / declare temp / node

traverse tree to find parent node

place in left / right child

Vocab

- **leaf node:** node in tree w/ no children
- **tree height:** number of edges between root to deepest leaf node
- **balanced tree:** minimum possible max. height (for each node, the heights of left/right subtrees of x differ by at most 1)

Chapter 10 Notes

→ **recursion:** process of a function calling itself
↳ frequently used to evaluate structures that can be defined by self-similarity
→ recursion must include:
↳ base case: smallest unit of problem that can be defined
↳ set of rules that reduce all cases to base case

Tree Traversal

→ **inorder:** left, parent, right
→ **preorder:** parent, left, right
→ **postorder:** left, right, parent

Chapter 11 Notes

Red-Black Trees

→ each node in BST is assigned red/black & nodes are ordered such that no path from the root to a leaf can be more than twice as long as any other path
→ height of O(log n)
→ struct includes color
→ root / leafs are black
→ if node is red, both of its children must be black
→ for each node in tree, all paths from that node to leaf nodes contain same number of black nodes

```
left Rotate (node x) // rotates subtree about node x in red-black tree
    node y = x->right
    x->right = y->left
    if(y->left != NULL)
        y->left->parent=x
    y->parent = x->parent
    IF(x->parent == NULL)
        root = y
    else
        if(x== x->parent->left)
            x->parent->left=y
        else
            x->parent->right=y
    y->left=x
    x->parent=y
```

```
rightRotate(node y)
  node x= y->left
  y->left= x->right
  IF(x->left != NULL)
     x->right->parent=y
  x->parent=y->parent
  IF(y->parent==NULL)
    root=x
  else if (y== (y->parent->left)
    y->parent->left=x
  else
     y->parent->right=x
  x->right=y
  y->parent=x
```

Chapter 12 Notes

Graphs

→ **graphs**: provide a structure for representing connections between people, places, or things that capture essence of connections

Adjacency Matrix

→ structure for representing direct connections between entities in a graph

→ 2D matrix - entities are listed on both horizontal/vertical axis

→ if connected, adjacent = 1 at location in matrix, else 0

Graph Representation

→ $G = (V, E)$: Graph G has set number of vertices connected by set of E edges

→ can represent info in adjacency matrix

Adjacency List Representation

→ vertices in graph are stored in an array & each vertex in the array contains a pointer to a list of adjacent vertices

Directed/Undirected Graphs

→ **undirected**: edge between 2 vertices exists in both directions

→ **directed**: edges between 2 vertices have a direction associated w/ them

Weighted Graphs

→ **weighted graph**: edge has a weight that provides info about the connection

Graph ADT

private:
- vertices

public:
- Init()
- insertVertex(val)
- insertEdge(startval, endval, weight)
- deleteVertex(val)
- deleteEdge(startval, endval)
- printGraph()
- search(value)

Implementation

→ uses vectors instead of arrays to simplify memory management
→ vector<type> variable

Traversal

→ BFS: search algorithm that evaluates nodes in breadth first ordering (root, left→rt, rt→lt)
→ depth-first search: evaluates vertices along one path before other paths

Chapter 13 Notes

Hash Tables

→ **hash table:** data structure that stores data using a parameter in the data to map the index in an array

→ necessary components: array where the records are stored and hash function that generates the mapping to an array index

Hash Functions

→ convert the key into an integer index to store in the hash table

→ simplest hash fcn: converts string to integer by summing ASCII values of all letters & modding the sum by array size

Collisions

→ multiple records always have the same hash value

↳ if $h(k_1) = h(k_2)$ & $(k_1 != k_2)$

Hash Functions

→ challenge of designing a good hash table is designing a hash fcn that limits the frequency of collision

→ must consider size & hash function

Perfect Hash Function

→ assigns all records to a location in hash table w/o collisions or wasted space

→ **imperfect fcn:** multiple keys can be assigned to the same index & result in collisions

Multiplication Method

→ category of hash fcns uses multiplication as part of fcn

Collision Resolution

→ use open addressing/chaining to handle collisions

→ open addressing: once collision has been identified, separate function is used to traverse through hash table array & place record in first available location

→ chaining: hash table is set up as an array of pointers that serve as the head of a linked list for a given hash value