

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 algorithm's 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 $n \rightarrow \infty$
 - ↳ $n = \text{size of program's input (any data bits)}$

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 contiguous block of memory
 - ↳ makes it easy and fast to access
- limitations:
 - ↳ fixed size (need array doubling algorithms for it)
- algorithm to delete array element
 - for $x = \text{index} \rightarrow \text{numElements} - 2$
 - $A[x] = A[x+1]$
 - $\text{numElements} = \text{numElements} - 1$
- 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
 - $B[x] = A[i]$
 - 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

bubble Sort(A)

for $i=0$ to $A.end - 1$

 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 move into correct location in the array

one at a time

insertion Sort(A)

for $i=1$ to $A.end$

 index = $A[i]$

 j = 1

 while ($j > 0 \wedge A[j-1] > index$)

$A[j] = A[j-1]$

 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
 - ↳ create another node dynamically & set the values for key, next and previous of n¹
 - ↳ connect the nodes
- traversing a linked list:
 - ↳ linked list nodes are accessed through the pointers stored in the list
- traverse()
 - temp = 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 = search(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 temp

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 and

↳ 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

→ Binary SearchTree

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

→ search iterative (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
node->rightChild = NULL

while (tmp != NULL)
    parent = tmp
    if (node->key < tmp->key)
        tmp = temp->leftChild
    else
        tmp = temp->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) // 1 child

 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.