Big-O notation

Counting basic operations, loops and recursion

\*\*Stacks

|  |
| --- |
| push/pop  FILO - First in last out  (==)LIFO - Last in first out  push(20),push(40), push(60), push(80)  0 20  1 40  2 60  3 80  4 0 <- top  public class myStack{  private int top;  private long[] stackArray;  private int maxSize;  public myStack(int size){  maxSize = size;  stackArray = new long[maxSize];  top = -1;  }    public void push(long value){  stackArray[++top] = value;  }    public long pop(){  return stackArray[top--];  }    public boolean isEmpty(){  return (top == -1)  }    public boolean isFull(){  return (top == maxSize)  } |

\*\*Queues

|  |
| --- |
| FIFO (a line)  front and rear  public class Queue{  private int maxSize;  private long[] queArray;  private int front;  private int rear;  public Queue(int size){  maxSize = size+1;  queArray = new long[maxSize];  front = 0;  rear = -1;  }  public void insert(long j){  if(rear==maxSize-1)  rear = -1;  queArray[++rear] = j;  }  public long remove(){  long temp = queArray[front++];  if(front == maxSize)  front = 0;  return temp;  }  public long peek(){  return queArray[front];  }  public boolean isEmpty(){  return (rear+1==front || (front+maxSize-1==rear));  }  public boolean isFull(){  return(rear+2==front || (front+maxSize-2==rear));  }  public int size(){  if(rear>=front)  return rear-front+1;  else  return (maxSize-front) + (rear+1);  } |

\*\*Priority Queues

Efficiency: Insert = O(1) to O(N)

Remove = O(N) or O(1)

\*\*Arrays

Linear and Binary Searching

\*\*Linked Lists

|  |
| --- |
| public void insertNode(Node n){  if(first==null){ //if empty  first = n;  }  else{  first.prev=n;  n.next=first;  first = n;  }  }  public void insertNth(Node n, int index, int key){  first = n; //start at specified node  Node newNode = new Node(key); //create node  Node tempCurrent = first; //start at front  if (index <= size - 1){  for(int i=index-1; i>0; i--){ //traverse index-1 amount of times  tempCurrent = tempCurrent.next;  }//end for  }//end if  newNode.next = tempCurrent.next;  tempCurrent.next = newNode;  size++;  }//end insertNth |
| public void insertEnd(Node n, int id){  first = n;  Node newNode = new Node(id);  if(isEmpty()){  first = newNode;  current = newNode;  current.next = null;  }//end if  else{ //not empty  current.next = newNode;  current = newNode;  current.next = null;  }//end else  size++;  }//end insertEnd |

\*\*Trees

* Search/Insert/Delete in O(logN) Time
* Each tree typically contains one path to each leaf.

|  |
| --- |
| *public node find(int key){*  *//assumes tree is not empty*  *Node current = root;*  *while(current.iData!=key){*  *if(key < current.iData) //key is less than current*  *current=current.leftchild*  *else //key is greater than current*  *current=current.rightchild*  *if(current=null)*  *return null;*  *}*  *return current;*  *}* |
| *public void insert(int id){*  *//assumes tree is not empty*  *Node newNode = new Node(id);*  *if(root == null)*  *root = newNode;*  *else{//tree not empty*  *Node current = root;*  *Node parent;*  *while(true){*  *parent = current; //trailing parent*  *if(id < current.iData){ //key is less than current*  *current=current.leftchild;*  *if (current==null){*  *parent.leftchild = newNode;*  *return;*  *}//end if*  *}//end if*  *else{ //key is greater than current*  *current=current.rightchild*  *if (current==null){*  *parent.rightchild = newNode;*  *return;*  *}//end if*  *}//end else*  *}//end while*  *}//end else*  *}//end insert* |

|  |  |
| --- | --- |
| \*\*In-order  *public void inOrder(Node root){*  *if (node == null)*  *return;*  *inOrder(root.leftChild());*  *System.out.print(root.iData() + “, ”);*  *inOrder(root.rightChild());*  *}//end inOrder* | \*\*Pre-order  *public void preOrder(Node root){*  *if (node == null)*  *return;*  *System.out.print(root.iData() + “, ”);*  *preOrder(root.leftChild());*  *preOrder(root.rightChild());*  *}//end preOrder* |
| \*\*Post-order  public void postOrder(Node root){  if(root==null)  return;  postOrder(root.leftChild());  postOrder(root.rightChild());  System.out.print(root.iData() + “, ”);  }//end postOrder | \*\*Level-order  public void recLevelOrder(Node root, int level) {  if (root == null)  return;  else if (level == 1)  System.out.print(root.iData + “ “);  else (level > 1) {  recLevelOrder(root.leftChild, level - 1);  recLevelOrder(root.rightChild, level - 1);  }//end else  }//end recLevelOrder |
|  | public void levelOrder(int height) {  for (int i = 1; i <= height; i++) {  recLevelOrder(root, i);  }//end for  }//end levelOrder  \*this one is above recLevelOrder^ |

\*\*Binary Search Trees

* + - A Binary Search tree has small left children, big right children for every non-leaf node
    - Deleting a node with one child from a binary search tree DOES NOT involve finding that node’s successor

**\*\*Heaps -** O(logN)

**Insertion**

|  |
| --- |
| public boolean insert(int key){  if(currentSize==MaxSize) //if array is full  return false;  Node newNode = new Node(key)  heapArray[currentSize] = newNode; //put new node at end  trickleUp(currentSize++);  return true;  } //end insert() |
| public void trickleUp(int index){  int parent = (index-1) / 2  Node bottom = heapArray[index];    while(index > 0 && heapArray[parent].getKey() < bottom.getKey()){  heapArray[index] = heapArray[parent]; //move node down  index = parent; //move index up  parent = (parent-1) / 2; //parent <- its parent  }//end while  heapArray[index] = bottom;  } //end trickleUp() |

**Removal**

|  |
| --- |
| public Node remove(){ //delete item with max key  Node root = heapArray[0]; //save the root  heapArray[0] = heapArray[--currentSize]; //root <- last  trickleDown();  return root;  } //end remove() |
| public void trickleDown(int index){  int largerChild;  Node top = heapArray[index]; //save root  while(index < currentSize /2){  int leftChild = 2\*index+1  int rightChild= leftChild+1  if(rightChild < currentSize &&  heap[leftChild].getKey() < heap[rightChild].getKey())  largerChild = rightChild; //find larger child  else  largerChild = leftChild;  if(top.getKey() >= heapArray[largerChild.getKey()) //top >= largerChild  break;  heapArray[index] = heapArray[largerChild]; //shift child up  index = largerChild; //go down  }//end while  heapArray[index] = top;  } |

**Key Change**

|  |
| --- |
| public boolean change(int index, int newValue){  if(index<0 || index>=currentSize)  return false;  int oldValue = heapArray[index].getKey(); //remember old  heapArray[index].setKey(newValue); //change to new    if(oldValue < newValue)  trickleUp(index);  else  trickleDown(index);  return true;  } //end change() |

**\*\*Hash Tables**

|  |
| --- |
| Efficiency: O(1) insertion/search/sometimes deletion  Open Addressing  Hash Function = make large range of numbers into smaller range  -arrayIndex = hugeNumber % arraySize;  Linear probing – identical indices are placed at x+1, x+2, x+3… (primary clustering)  Quadratic probing – identical indices are placed at x+1, x+4, x+9, x+16… (secondary clustering)  Double Hashing – Hash key second time, using a diff hash function, use result as step size.  -stepSize = constant – (key % constant); //where constant is prime && < arraySize  Separate Chaining – create linked list at each index in the hash table. Duplicates are added to end of linked list |
| Linear Probing:  public void insert(int key, String value){ //string so we don't accidentally use it as an index  int hash val = hashFunc(key); //assign key to hash value  int index = hashval % size; //compress that to the size of the array (assume the size is available)  //check if index is available  int start = index;  boolean full = 0;  while(hashTable[index] != null){ //finds spots  index++;  if(index == size)  index = 0; //wrap around  //if it's completely full  if(start == index){  full = 1;  break;  }  }  if(!full)  hashTable[index] = value;  }  }  public DataItem delete(int key){  int hashVal = hashFunc(key); //hash the key  while(hashArray[hashVal].getKey() == key){  if(hashArray[hashVal] != null){ //until empty cell  DataItem temp = hashArray[hashVal];  hashArray[hashVal] = nonItem;  return temp;  }  ++hashVal; //go to next cell  hashVal %= arraySize //wrap around if necessary  }  return null;  } |
| Double Hashing:  public void insert(int key, DataItem item){  int hashVal = hashFunc1(key); //hash key  int stepSize = hashFunc2(key); //get step size  while(hashArray[hashVal] != null &&  hashArray[hashVal.getKey() != -1){  hashVal += stepSize; //add the step  hashVal %= arraySize; //for wraparound  }  hashArray[hashVal] = item;  }  public DataItem delete(int key){  int hashVal = hashFunc1(key);  int stepSize = hashFunc2(key);  while(hashArray[hashVal] != null){  if(hashArray[hashVal].getKey() == key){  DataItem temp = hashArray[hashVal];  hashArray[hashVal] = nonItem;  return temp;  }  hashVal += stepSize  hashVal %= arraySize  }  return null;  } |
| Separate Chaining: |

Sorting Algorithms

\*Bubble Sort

|  |
| --- |
| public static void Bubble(int[] arr){  /\* BubbleSort invariant: During each pass; the largest number is put at the  \* rightmost position that hasn't been used.  \*/  int temp = -1;  boolean flag = true;  while(flag){  flag = false;  for(int i=1; i<arr.length; i++){  if(arr[i-1]>arr[i]){  temp = arr[i]; //store smaller in temp  arr[i] = arr[i-1]; //update smaller numbers index to bigger number  arr[i-1] = temp; //put smaller number in previous index  flag = true; //swap occurred  }  }  }  } |

\*Selection Sort

|  |
| --- |
| Invariant: left side of next element to be sorted is always sorted (in the final resting position) |
| public static void selectionSort(int[] arr) {  int i, j, first, temp;  //traverse array swapping smallest value with front index of the array  for(i=0; i<arr.length-1; i++){  first = i; //update front of array as we go (beginning part will be sorted)  //traverse the array beyond i each time to find the min in the unsorted array  for(j=i+1; j<arr.length; j++){  if(arr[j] < arr[first]) //if the value found is smaller than first item looked at  first = j; //update first to the index of smallest item  }//end for  temp = arr[first]; //swap first (smallest item) with current starting point (arr[i])  arr[first] = arr[i];  arr[i] = temp;  }  } |

\*Insertion Sort

|  |
| --- |
| public static void InsertionSort(int[] num){  int j;  for (int i = 1; i < num.length; i++) // Start with 1 (not 0)  {  int key = num[i];  for(j=i-1;(j >= 0) && (num[j]>key); j--){ // Smaller values are moving up  num[j+1] = num[j];  }  num[j+1] = key; // Put the key in its proper location  }  } |

\*Merge Sort

Efficiency: 0(NlogN)

Copies: 8log2(8)=24, NlogN

Compares: N-1 worst case, N/2 best case

Description: Divide array in half recursively until

only one item exists in the array (sorted), merge

pieces back together in order.

|  |
| --- |
| private void recMergeSort(long[] workSpace, int lBound, int uBound){  if(lBound == uBound) // range = 1,  return; // base case  else{  int mid = (lBound+uBound)/2; //find midpoint  recMergeSort(workSpace, lBound, mid);  recMergeSort(workSpace, mid+1, uBound);  mrege(workSpace, lBound, mid+1, upperBound);  }  } |
| private void merge(long[] workSpace, int lowPtr, int highPtr, int uBound){  int j = 0;  int lBound = lowPtr;  int mid = highPtr - 1;  int n = upperBound-lowerBound+1 //# of items  while(lowPtr<=mid && highPtr<=uBound){  if(theArray[lowPtr] < theArray[highPtr])  workSpace[j++] = theArray[lowPtr++];  else  workSpace[j++] = theArray[highPtr++];  }  while(lowPtr<=mid)  workspace[j++] = theArray[lowPtr++];  while(highPtr<=uBound)  workspace[j++] = theArray[highPtr++];  for(j=0; j<n; j++)  theArray[lowerBound+j] = workSpace[j];  } |

\*Quick Sort

|  |
| --- |
| partition algorithm  O(N), N+1 or N+2 comparisons, < (N/2) swaps  Works by starting with two pointers, one at each end of the array.  When leftPtr encounters a data item > pivot value, it stops, ready to swap  When rightPtr encounters a data item < pivot value, it stops, ready to swap  then swap(leftPtr, rightPtr); |
| Efficiency:  worst case - n-1 times, so called worst case = O(n)\*O(n-1) = O(n^2)  best case - log n, so called best case = O(n)\*O(logn), so O(nlog n)  Partition array into smaller keys and larger keys of a chosen pivot  recurse on left group  recurse on right group  picking a pivot besides array[right]  median of 3: pick first, middle, and last num in array  compare all 3 and pick middle value |

\*Shell Sort

|  |
| --- |
| Efficiency: O(N^(3/2)) down to O(N^(7/6)), [O(nlogn)< X < O(n^2)]  Invariant: everything on left of current is partially sorted  Description: insertion sort on widley spaced elements, spaced by an increment  after the increment finishes, it decreases based on knuth's algorithm  h = 3\*h+1 (increasing) 1,4,13,40,121,364  h = (h-1)/3 (decreasing) |

**\*Heap Sort**

Efficiency: O(NlogN), though slower than quicksort because of trickleDown

|  |
| --- |
| After creating a heap:  for(j=size-1; j>=0; j--){  Node biggestNode = theHeap.remove();  theHeap.insertAt(j, biggestNode);  } |

\*Radix Sort

|  |
| --- |
| Efficiency: O(Nlogn)  for each item, copy to appropriate linked list - n and n, O(2n) = O(n)  + then back to the array  repeat for each digit 3 (# of digits)\*2n (# of copies) = O(n)  O(K\*n) K = # of digits or O(NlogN) (K=log2N)    disassembles "Keys" (number we want to sort) into digits & arranges data items according to value of each digit  \*no comparisons  1) Take all items and divide them into 10 groups based on 1's digit (0 to 9)  2) Reassemble groups: all keys ending in 0 -> all keys ending in 1, 2, 3..., 9  3) Repeat this for every digit (10's digit, 100s digit, etc)  \*4) must be done w/o changing the order of the previous step |

Graphs

\*\*Graph Representation (Adjacency List, Adjacency Matrix)

\*Breadth-first search (BFS)

|  |
| --- |
| USES QUEUE/Stay close  RULES:  (0) define start pt  (1) visit next unvisited adjacent vertex, mark it, insert into queue  (2) if you can't follow rule 1, remove vertex from queue, + MARK IT "Current"  (3) if you can't follow 1 or 2, you are done |

\*Depth-first search (DFS)

|  |
| --- |
| USES STACK/Explore whole path  RULES:  (0) define start pt, put on stack  (1) if possible, visit adjacent vertex, mark it True, put it on the stack  (2) if you can't follow rule 1, if possible, pop vertex (don't change boolean)  (3) if you can't follow 1 or 2, you are done |

\*Minimum Spanning Trees

Minimizes # of edges between vertices in a given graph

\*Shortest Path

Dykstra’s Algorithm – one specified vertex to all other vertices

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | B | C | D | E |
| A | 10 | 5 | inf | inf |
| C | 5+9 | X | inf | 5+1 |
| E | X | X | 5+1+3 | X |
| D | X | X | X | X |
| B | X | X | X | X |

**\*Transitive Closure**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 1 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 1 | 1 |
| C | 1 | 1 | 1 | 1 | 1 |
| D | 1 | 1 | 1 | 1 | 1 |
| E | 1 | 1 | 1 | 1 | 1 |

**\*if a node can reach another node = 1, else 0**

\*Hamiltonian Cycles